# iAPX 386 ARCHITECTURE SPECIFICATION

TABLE OF CONTENTS

1 Introduction .................................... 1

2 Programmer Model .................................... 2
2.1 Data Types .................................... 2
2.1.1 Bit String Data Type .................................... 3
2.2 Register Model .................................... 4
2.2.1 General Registers .................................... 5
2.2.2 Segment Registers .................................... 6
2.2.3 Special Registers .................................... 7
2.2.4 Control Registers .................................... 10
2.2.4.1 CR0: Machine Status Register .................................... 10
2.2.4.2 CR1: (RESERVED) .................................... 12
2.2.4.3 CR2: Page Fault Linear Address .................................... 12
2.2.4.4 CR3: Page Directory Base .................................... 12
2.2.5 System Address Registers .................................... 13
2.3 Operand Addressing .................................... 17
2.3.1 Immediate Operands .................................... 17
2.3.2 Register Operands .................................... 17
2.3.3 Memory Operands .................................... 18
2.3.3.1 Effective Address Computation .................................... 18
2.3.3.2 16-Bit Addressing Modes .................................... 18
2.3.3.3 32-Bit Addressing Modes .................................... 20
2.4 Data Alignment .................................... 25

3 PROTECTED MODE MEMORY ADDRESSING .................................... 26
3.1 Linear Address Formation .................................... 26
3.2 SELECTOR FORMAT .................................... 26
3.3 DESCRIPTOR FORMATS .................................... 28
3.3.1 Memory Segments .................................... 28
3.3.1.1 Data Segments .................................... 29
3.3.1.2 Code Segments .................................... 32
3.3.2 Special Memory Segments .................................... 33
3.3.2.1 TYPE = 2, Local Descriptor Table .................................... 34
3.3.2.2 TYPE = 1, 3 286 Task State Segment .................................... 34
3.3.2.3 TYPE = 9, 11 386 Task State Segment .................................... 37
3.3.3 Gates .................................... 40
3.3.3.1 TYPE = 4, 286 Call Gate .................................... 40
3.3.3.2 TYPE = 5, Task Gate .................................... 41
3.3.3.3 TYPE = 6, 286 Interrupt Gate .................................... 41
3.3.3.4 TYPE = 7, 286 Trap Gate .................................... 41
3.3.3.5 TYPE = 12, 386 Call Gate .................................... 41
3.3.3.6 TYPE = 14, 386 Interrupt Gate .................................... 42
3.3.3.7 TYPE = 15, 386 Trap Gate .................................... 42
3.4 PAGING .................................... 43
3.4.1 Linear Address Space Partitioning .................................... 43
3.4.2 Page Table Structure .................................... 45
3.4.3 Directory and Page Table Entries .................................... 46
3.4.3.1 Present Bit .................................... 47
3.4.3.2 Page level Protection (U/S, R/W Bits) .................................... 47
3.4.3.3 Page Usage Statistics (D, A Bits) .................................... 51

4 INTERRUPTS AND EXCEPTIONS ........ 53
4.1 INTERRUPTS ........ 53
4.2 EXCEPTIONS ........ 55
4.3 PRIORITIZING INTERRUPTS AND EXCEPTIONS ........ 56
4.3.1 Enabling/Disabling ........ 56
4.3.2 Simultaneous Priority ........ 57
4.4 INTERRUPT/EXCEPTION SIGNALLING ........ 58
4.4.1 REAL mode Signalling ........ 58
4.4.2 Protected Mode Signalling ........ 59
4.4.3 Virtual 8086 Signalling ........ 60
4.5 PAGE FAULT HANDLING ........ 61
4.5.1 Page Fault Signalling ........ 61
4.5.1.1 Error Code Format ........ 61
4.6 SYSTEM ERRORS ........ 63
4.7 Restartability Implications on Software ........ 65
4.7.1 Partial Modification of Machine State ........ 65
4.7.2 Aliasing of Operands ........ 66
4.7.3 Stack Areas ........ 66

5 EXECUTING 8086 CODE ........ 68
5.1 Introduction ........ 68
5.2 8086 MODE ADDRESSING ........ 69
5.3 UNDEFINED OPCODE TRAPS ........ 71
5.4 PRIVILEGED AND SENSITIVE INSTRUCTIONS ........ 71
5.5 ENTERING AND LEAVING REAL MODE ........ 72
5.6 ENTERING AND LEAVING VIRTUAL 8086 MODE ........ 72
5.6.1 Task Switches to/from Virtual 8086 ........ 73
5.6.2 Transitions Through Trap and Interrupt Gates, and IRET ........ 73

6 Instruction Set ........ 76
6.1 86 FAMILY EXTENSIONS ........ 77
6.1.1 Segment Register Instructions ........ 77
6.1.1.1 Segment Register Override Prefixes ........ 78
6.1.1.2 Segment Register Move Instructions ........ 79
6.1.1.3 Pointer Loads LSS, LFS, LGS ........ 80
6.1.1.4 Push/Pop Segment Register ........ 81
6.1.2 Uncharacterized Multiply ........ 82
6.1.3 Move With Sign/Zero Extend ........ 83
6.1.4 Long Displacement Conditional Jumps ........ 85
6.1.5 BYTE Set on Condition ........ 85
6.1.6 Single Bit Operations ........ 87
6.1.6.1 Bit Test ........ 88
6.1.6.2 Bit Test and Set ........ 88
6.1.6.3 Bit Test and Reset ........ 89
6.1.6.4 Bit Test and Complement ........ 89
6.1.7 Double Shift Instructions ........ 90
6.1.7.1 Shift Left Double ........ 91
6.1.7.2 Shift Right Double ........ 92
6.1.8 INSERT and EXTRACT ........ 94
6.1.8.1 INSERT base, offset, length, source ........ 96
6.1.8.2 EXTRACT dest, base, offset, length ........ 98

RESTRICTED CONFIDENTIAL

6.1.9 ScanBit .......................................... 100
6.1.9.1 ScanBitForward Dest, Src ................. 100
6.1.9.2 ScanBitReverse Dest, Src ................. 102
6.2 386 PROCESSOR SPECIFIC EXTENSIONS ............. 103
6.2.1 Load/Store Control Registers ................ 103
6.2.2 Load/Store Debug Registers .................. 104
6.2.3 Load/Store Test Registers ................... 105
6.3 INSTRUCTION ENHANCEMENT SUMMARY ............... 106
6.4 32-BIT EXTENSIONS ............................. 109
6.4.1 DATA MOVEMENT INSTRUCTIONS .................. 110
6.4.1.1 Move Instructions ......................... 110
6.4.1.2 Segment Register Instructions ............. 112
6.4.1.3 I/O Instructions .......................... 113
6.4.1.4 Flags Instructions ........................ 115
6.4.2 ARITHMETIC/LOGICAL INSTRUCTIONS ............. 116
6.4.2.1 Standard Arith/Logical Instructions ....... 116
6.4.2.2 Multiply and Divide ....................... 119
6.4.2.3 Conversion Instructions ................... 121
6.4.2.4 Decimal Arithmetic Instructions ........... 122
6.4.2.5 Coprocessor Instructions .................. 123
6.4.3 STRING INSTRUCTIONS ......................... 126
6.4.4 CONTROL TRANSFERS ........................... 128
6.4.4.1 Procedure Call and Return ................. 130
6.4.4.2 Jump Instructions ......................... 132
6.4.4.3 Interrupt Instructions .................... 133
6.4.5 PROTECTION MODEL INSTRUCTIONS ............... 135
6.4.5.1 LIDT, LGDT, SIDT, SGDT .................... 135
6.4.5.2 LAR ....................................... 136
6.4.5.3 LSL ....................................... 137
6.4.6 MISCELLANEOUS INSTRUCTIONS .................. 138

7 Debug Support ................................... 140
7.1 Terminology ................................... 140
7.2 Debug Registers ............................... 141
7.2.1 Debug Conditions ............................ 144
7.2.2 Breakpoint Address Recognition .............. 145
7.3 System Errors ................................. 146
7.4 Programming Notes ............................. 146

APPENDIX E: Summary of 32-bit Extensions ............. 148

# 1. Introduction

This document describes a P3 extension to the 86/186/286 family architecture to include full 32-bit support. Software migration, or the portation of an existing 86/186/286 application to the P3, is supported by having the architecture bit-compatible with the existing 86 family architecture. Software upgrades, or the transformation of a migrated system to use new 32-bit features, are supported by structuring the architectural extensions to preserve the mechanisms provided on the 286.

(1)

# 2. Programmer Model

## 2.1. Data Types

In addition to the data types supported by the 86 family architecture, 32 bit signed and unsigned integers, and bit fields from 1 to 32 bits in length are supported. In addition to the pointer types supported by the 8086 family, the 386 supports a 32-bit offset-only pointer, and a 48-bit full pointer.

On the 386, as well as all other members of the 86 family of processors, a WORD is 16-bits wide, and a DWORD is 32 bits wide. WORDs are stored in two consecutive bytes in memory with the low-order byte at the lowest address, the high order byte at the high address. DWORDs are stored in four consecutive bytes in memory with the low-order byte at the lowest address, the high-order byte at the highest address. The address of a WORD or DWORD data item is the byte address of the low-order byte.

Arithmetic:

- 8, 16, 32-bit Integers
- 8, 16, 32-bit Ordinals
- 2-digit BCD
- 1-digit ASCII decimal

Floating-Point (supported in co-processor):

- 16, 32, 64-bit Integers
- 32, 64, 80-bit IEEE standard formats
- 80-bit decimal format

Pointer:

- 16-bit selector
- 16, 32-bit offset-only (SMALL pointer)
- 32, 48-bit full pointer

String

- variable length BYTE string (0..4G bytes in size)
- variable length WORD string (0..4G bytes in size)
- variable length DWORD string (0..4G bytes in size)

Bit String

- variable length bit string (1..32 bits)

### 2.1.1. Bit String Data Type

The 386 implements a set of extensions for manipulating bits and bit strings. Two kinds of instructions are provided. One set operates on a single bit within a large bit string. The second set operates on a sequence of up to 31 bits within a larger bit string.

A Bit String may be stored in a general register or in memory. The function

BIT[Base, Offset]

is defined to address a bit within a bit string. Base can be a general register or a memory address.

Bits are numbered from right to left within registers and memory bytes. The low order bit is bit 0, the high order bit is bit 7 (for memory bytes and Byte registers), 15 (Word Registers), or 31 (32-bit Registers). Bits are numbered in this fashion so that the bit numbers of words in memory are consistant with bit numbers of the same word in a register. In memory, the two bytes of a word are stored with the low order byte at the lower address, which contains the lower numbered bits of the word. The byte-swapping on the 386 implies a low-to-high (or right-to-left) numbering of bits.

If the Base operand is a register, the offset operand can be in the range 0..31. This offset addresses a bit within the indicated register.

```
 3                   2
 1                   1                                      0
+--------------------+-+------------------------------------+
|                    | |                                    |
|                    | |                                    |
|                    | |                                    |
+--------------------+-+------------------------------------+
                     A                                      |
                     |------------ OFFSET = 21 -------------|
```

Register Bit Indexing

If the base operand is a memory address, the bit offset may range from -2Gigabits to 2 Gigabits. The addressed bit is numbered (Offset MOD 8) within the byte at address (Base + (Offset DIV 8)), where DIV is signed division with rounding towards negative infinity, and MOD returns a positive number.

IMPLMENTATION NOTE:

> The bit offset within the byte is simply the low order three bits of the Offset operand, zero extended on the left. The Byte offset is the upper 29 bits (13 for 16-bit operand size), <u>sign</u>-extended to 32 bits (16 bits).

For example, the following figure illustrates a bit offset of 13.

```
 7 6 5 4 3 2 1 0 7 6 5 4 3 2 1 0 7 6 5 4 3 2 1 0 7 6 5 4 3 2 1 0
+---------------+---------------+---------------+---------------+
|   | |         |               |               |               |
|   | |         |               |               |               |
|   | |         |               |               |               |
+---------------+---------------+---------------+---------------+
|               |               |               |               |
|  Base+1       |    Base       |   Base-1      |   Base-2      |
                                |
     A                          |
     |------- Offset = 13 ------|
```

Memory Bit Indexing (positive offset)

As a final example, the following figure illustrates a bit offset of -11 from the same base memory address.

```
 7 6 5 4 3 2 1 0 7 6 5 4 3 2 1 0 7 6 5 4 3 2 1 0 7 6 5 4 3 2 1 0
+---------------+---------------+---------------+---------------+
|               |               |               |   | |         |
|               |               |               |   | |         |
|               |               |               |   | |         |
+---------------+---------------+---------------+---------------+
|               |               |               |               |
|  Base+1       |    Base       |   Base-1      |   Base-2      |
|                               |
                                |                    A
                                |                    |
                                |--- Offset = -11 ---|
                                |                    |
```

Memory Bit Indexing (negative offset)

## 2.2. Register Model

The register model extends the 86/186/286 general registers (AX, BX, CX, DX, SI, DI, SP, BP) to 32 bits by adding 16 high-order bits. These extended registers can be accessed as 16 or 32-bit registers (the X registers can also be accessed as 8-bit registers).

Two segment registers, FS and GS, are added to alleviate the congestion in the ES register, and to better match the base and index registers available in the general register set. The segment registers remain 16 bits wide.

### 2.2.1. General Registers

There are 8 general registers of 32 bits named EAX, ECX, EDX, EBX, ESP, EBP, ESI, and EDI. These registers have byte access in the first four registers at the low order word of each register: AL .. BH. There is low order word access and double word access to all registers. When accessed as 8 or 16-bit registers, the upper bits are undisturbed.

GENERAL REGISTER SET

```
+-----------------+-----------------+
|                 |       AX        |
|                 |                 |    EAX
|                 |   AH   |   AL   |
+-----------------+--------+--------+
|                 |       CX        |
|                 |                 |    ECX
|                 |   CH   |   CL   |
+-----------------+--------+--------+
|                 |       DX        |
|                 |                 |    EDX
|                 |   DH   |   DL   |
+-----------------+--------+--------+
|                 |       BX        |
|                 |                 |    EBX
|                 |   BH   |   BL   |
+-----------------+--------+--------+
|                 |                 |
|                 |       SP        |    ESP
|                 |                 |
+-----------------+-----------------+
|                 |                 |
|                 |       BP        |    EBP
|                 |                 |
+-----------------+-----------------+
|                 |                 |
|                 |       SI        |    ESI
|                 |                 |
+-----------------+-----------------+
|                 |                 |
|                 |       DI        |    EDI
|                 |                 |
+-----------------+--------+--------+
```

The register codes for the general registers as used in instructions and the 32-bit addressing modes are given in the section 2.3.2. The register selected depends on the operand length of the instruction. Only the 32-bit registers can be used in the 32-bit address modes.

### 2.2.2. Segment Registers

There are 6 segment registers of 16 bits named ES, CS, SS, DS, FS, and GS.

SEGMENT REGISTERS

```
+-----------------+
|                 |
|                 |   ES
|                 |
+-----------------+
|                 |
|                 |   CS
|                 |
+-----------------+
|                 |
|                 |   SS
|                 |
+-----------------+
|                 |
|                 |   DS
|                 |
+-----------------+
|                 |
|                 |   FS
|                 |
+-----------------+
|                 |
|                 |   GS
|                 |
+-----------------+
```

The segment registers are encoded with register numbers given in the following table.

| SReg Code | Register Name |
|---|---|
| 000 | ES |
| 001 | CS |
| 010 | SS |
| 011 | DS |
| 100 | FS |
| 101 | GS |

### 2.2.3. Special Registers

The Extended Instruction Pointer (EIP) is a 32 bit register used as an instruction counter. The lower 16 bits of the EIP register contain the 86/186/286 Instruction Pointer (IP) register. IP was extended to support the expansion of code segments beyond the 64K size in the 8086 and 286.

The FLAGS register is extended to 32 bits, with a few new bits defined in the upper 16 bits. New bits were added by extending the FLAGS register rather than using undefined low order bits, in order to minimize compatiblility problems.

```
                    SPECIAL REGISTERS

 31                15                0
 +-----------------+-----------------+
 |                 |                 |
 |                 |      FLAGS      |  EFLAGS
 |                 |                 |
 +-----------------+-----------------+


 +-----------------+-----------------+
 |                 |                 |
 |                 |       IP        |   EIP
 |                 |                 |
 +-----------------+-----------------+
```

```
                       FLAG REGISTER DETAILS
 3 3 2 2 2 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1 1
 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0
+---------------+---------------+-------+-----------------------+
|               |            V R|x N IOP|O D I T S Z x A x P x C|
|0 0 0 0 0 0 0 0|0 0 0 0 0 0 M F|0 T  L |F F F F F F 0 F 0 F 1 F|
+---------------+---------------+-------+-----------------------+
```

Flag bits 1, 3, 5, 15, and 31.18 are "undefined", but when stored with SAHF or PUSHF or Interrupt processing will store a 1 in bit 1, and zeros in bits 3, 5, 15, and 31..18.

COMPATIBILITY NOTE:

Bits 1, 3, and 5 are stored with these values to be compatible with the 8080 flags, which were carried over to the 8086. The setting of the flags stored with PUSHF and stored by interrupts will be different from those stored on the 8086 in bits 12..15. On the 8086 these bits were stored as 1111, but in 286 REAL mode bit 15 is always zero, and bits 14..12 will reflect the last value popped into them.

The defined flag bits are interpreted as follows:

VM The VM bit provides Virtual 8086 Mode. If set to 1 while the processor is executing in Protected Mode, the machine will switch to Virtual 8086 mode. Set only by IRET (if CPL=0) and task switches in Protected Mode (at any CPL). Cleared by interrupts (at any CPL). Unaffected by POPF. PUSHF always pushes a 0 in this bit, even if processor is executing in Virtual 8086 Mode. FLAGS image pushed during interrupt processing, or saved during task switches, will contain a 1 in this bit if the interrupted process was executing in Virtual 8086 Mode. See Chapter 5 for more details.

RF is reset at the successful completion (no faults are signalled) of every instruction except for IRET, POPF, and instructions which perform a task switch. The RF flag is set only by these instructions, which will set RF from the popped/loaded FLAGS image before the break processing for the next instruction boundary is done.

The RF bit is set in the flags image stored as part of processing a fault (but not traps or interrupts).

If RF is set at an instruction boundary, debug faults are disabled.

RF is used to easily resume program execution after processing a debug fault. Because debug faults resume with the instruction that faulted, without the RF bit, simply returning to the faulting instruction with the fault condition still enabled would cause the processor to spin reporting the same fault on the same instruction. Furthermore, if a breakpointed instruction got one or more page faults, multiple breakpoints would be reported for the "same" execution of the breakpointed instruction: each retry from the page fault handler would report another breakpoint. The RF bit ensures that only one breakpoint is reported.

IMPLEMENTATION NOTE:

"Reset at successful completion" is equivalent to reset at beginning combined with being set by fault handlers.

NT Nested Task Flag. This bit is set to 1 to indicate that the execution of this task is "nested" within another task. If 1, it indicates that the current task's TSS has a valid back

| | |
|---|---|
| | link to the previous TSS. This bit is set or reset by control transfers through interrupt, trap, and task gates. Its value is tested by the IRET instruction to determine whether to do an inter-task return or an intra-task return. A POPF or IRET instruction *will* affect the setting of this bit at *any* privilege level. Task switches will always alter this field. |
| IOPL | The I/O Privilege Level field indicates the maximum privilege level permitted to perform I/O instructions. POPF and IRET instructions will alter this field only at level 0. Task switches will always alter this field. |
| OF | is set if the operation resulted in signed overflow, that is, if the operation resulted in a carry/borrow *into* the high-order bit of the result, but not a carry/borrow *out of* the high-order bit, or vice-versa. For 32-bit operations OF is set according to the overflow at bit 31. |
| DF | This is the direction flag used by the string instructions. |
| IF | The Interrupt enable Flag can be used to enable/disable certain classes of external interrupts. Refer to Chapter 5.<br>This flag is not affected by a POPF or IRET instruction except at privilege levels <= IOPL. It is always affected by a task switch. |
| TF | The Trap Enable Flag controls the generation of Single-Step interrupts. Refer to chapter 4 and 7. |
| SF | is set if the high-order bit of the result is set; otherwise SF is cleared. For example, 32-bit operations set SF to the value of bit 31 of the result. |
| ZF | is set if the result is zero, otherwise cleared. For example, 32-bit operations set ZF if a 32-bit zero result is produced. |
| AF | is set if the operation resulted in a carry our of (from addition) or borrow out of (from subtraction) the low-order 4 bits of the result; otherwise AF is cleared. AF is set according to the carries/borrows out of bit 3, and is not affected by the operand size (8, 16, 32-bit operations). |
| PF | is set if the modulo 2 sum of the low-order eight bits of the result of the operation is 0 (even parity); otherwise PF is cleared (odd parity). The setting of PF is not affected by operand size (8, 16, 32-bit operations set PF according to the low-order 8 bits of the result). |
| CF | is set if the operation resulted in a carry out of (from addition) or a borrow out of (from subtraction) the high- |

order bit of the result; otherwise CF is cleared. For example, 32-bit operations set CF according to the carries/borrows out of bit 31.

### 2.2.4. Control Registers

The 386 defines a set of 4 32-bit control registers to hold additional (over the 286) global machine state. These registers, along with the System Address Registers described in the next section, hold machine state that affects all tasks in the system. Two instructions, load and store, are defined in chapter 6 to provide access to the control register set. The 286 MSW register is contained in the low order 16 bits of CR0. The 286 LMSW and SMSW instructions are taken as special aliases of the load and store CR0 instructions. Each control register is described in detail below.

```
 3             2           1 1 1 1     1 1
 1             4           8 7 6 5     2 1     8 7     4 3 2 1 0
+-+-------------+---------------+---------------+-----+-+-+-+-+-+
|P|                             |                     |E|T|E|M|P|
| |        RESERVED             |                     | | | | | |    CR0|
|G|                             |                     |T|S|M|P|E|
+-+-------------+---------------+---------------+-----+-+-+-+-+-+
|                                                               |
|                             UNUSED                            |    CR1
|                                                               |
+---------------------------------------------------------------+
|                                                               |
|              PAGE FAULT LINEAR ADDRESS                        |    CR2
|                                                               |
+---------------------------------------+-----------------------+
| PAGE DIRECTORY BASE REGISTER (PDBR)   |                       |
|                                       |0 0 0 0 0 0 0 0 0 0 0 0|    CR3
| PHYSICAL ADDRESS, PAGE ALIGNED        |                       |
+---------------------------------------+-----------------------+
```

PROCESSOR CONTROL REGISTERS

#### 2.2.4.1. CR0: Machine Status Register

The MSW register of the 286 is mapped into the low-order halfword of CR0. Loading CR0 has no effect on the contents of any internal caches.

PG The PaGing enable bit (bit 31, not in the 286) is used to indicate whether paging is enabled or not.

| | |
|---|---|
| ET | This bit indicates the processor Extension Type. It is 0 if a 287 coprocessor is in the system. It is a 1 if a 387 coprocessor is in the system. If 1, the 32-bit coprocessor protocol is used to provide higher performance. If 0, the 16-bit, 287 compatible coprocessor protocol is used. |
| TS | The Task Switched bit is set whenever a task switch operation is performed. If it is set, execution of a coprocessor ESC instruction will cause a Device Not Available Fault (trap vector 7). |
| EM | The EMulate coprocessor bit determines whether ESC instructions are trapped (EM=1) or are sent to a coprocessor. If EM=1, all ESC instructions will raise a Device Not Available Fault (trap vector 7). Note that WAIT instructions are not affected by the EM bit's setting. |
| MP | The Monitor coProcessor bit is used in conjunction with the TS bit to determine if WAIT instructions should trap. If MP=1 AND TS=1, WAIT instructions will cause a Device Not Available Fault (trap vector 7). Note that ESC instructions are not affected by the TS bit's setting. Also note that on a task switch, the TS bit is set to 1 by the processor. |
| PE | The Protection Enable bit (bit 0) is set to enable the segment based protection mechanism. If PE=0, the processor operates in REAL mode, with segment based protection disabled, and addresses formed as in an 8086. |

COMPATIBILITY NOTE:

The new CR0 bits, except for ET, are upwards compatible from the 286, since they are in register bits not defined on the 286. For strict compatibility with the 286, the LMSW instruction is defined to not change the ET bit.

The PG and PE bits have 4 combinations, 3 of which define valid operating modes of the 386:

| PG | PE | MODE |
|---|---|---|
| 0 | 0 | REAL Mode. Exact 8086 semantics, with 32-bit extensions available with prefixes. See chapter 5. |
| 0 | 1 | Protected Mode. Exact 286 semantics, plus 32-bit extensions through both prefixes and "default" prefix setting associated with code segment descriptors. Also, a sub-mode is defined to support a virtual 8086 within the context of the extended 286 protection model. |

| | | |
|---|---|---|
| 1 | 0 | UNDEFINED. Loading CR0 with this combination of PG an PE bits will raise a GP fault with error code 0. |
| 1 | 1 | Paged Protected Mode. All the facilities of Protected mode, with paging enabled underneath segmentation. |

PROGRAMMING NOTE:

None of the caches are flushed when Control Register 0 (MSW) is loaded. Before paging is enabled by loading CR0, it is reccommended that these caches be flushed by other instructions preceding the load of CR0.

COMPATIBILITY NOTE:

The 286 initializes the MSW register to FFF0. The 386 will initialize this register to 0. The 286 defines only the low order 4 bits, which are initialized to 0. Programs which read the value of the MSW, and depend on having the unused upper bits set to 1 will behave differently on the 386.

PROGRAMMING NOTE:

The PE bit is sticky on the 286 - once set it cannot be reset. This is carried forward to the 386 by having LMSW behave as on the 286, with PE sticky. However, MOV CR0, R/M can be used to load the full register with new values, in this case potentially disabling protection and/or paging. Care must be taken to be sure that if paging is disabled, the processor is executing in a page(s) that has its linear address equal to its physical address. Disabling protection by loading a 0 into the PE bit in CR0 (assuming the current value is 1) will have unpredictable effects. Loading CR0 with PE=0 and PG=1 will raise a GP fault with error code 0. It is reccommended that operating system software exercise care when loading CR0.

### 2.2.4.2. CR1: (RESERVED)

### 2.2.4.3. CR2: Page Fault Linear Address

This register holds the full linear address which caused the last page fault detected. The error code pushed when the page fault handler is invoked passes additional status information on this page fault.

### 2.2.4.4. CR3: Page Directory Base

CR3 contains the physical page base address for the first level of the page table structure. The lower 12 bits of this register are ignored on loads, and have undefined values when stored. This forces the first level page table to be on a page boundary.

A task switch through a TSS which changes the value in this register, or an explicit load of CR3 with any value, will invalidate cached page table entries from the previous address space.

This will also invalidate any on-board caches whose tags depend on virtual addresses. A task switch through a TSS which has the same value in the CR3 image is not required to flush the caches.

PROGRAMMING NOTE:

> The Page Directory base address in CR3 is a physical address. The Page Directory can be paged out while its associated task is suspended, but the OS must insure that the Page Directory is resident in physical memory before the task is dispatched. The entry in the TSS for CR3 has a physical address, with no provision for a present bit. This means that the Page Directory for a task must be resident in physical memory, and the CR3 image in a TSS pointed to this area, before the task can be dispatched through its TSS.

### 2.2.5. System Address Registers

Four special segments are defined to hold the tables which describe the objects supported by the 286/386 protection model. These are the GDT (Global Descriptor Table), the LDT (Local Descriptor Table), the IDT (Interrupt Descriptor Table), and the TSS (Task State Segment). The addresses of these segments are stored in special registers, called the System Address Registers. Two of the segments are identified by segment selectors stored in special registers. The TR holds the TSS selector, and the LDTR holds the LDT selector. The GDT and IDT segments are defined by absolute base addresses and limit values rather than with selectors. These base and limit values are stored in 48-bit registers GDTR (GDT base and limit) and IDTR (IDT base and limit). The GDT must be specified by linear address values, since it is the root of the collection of segmentation tables. The IDT is also specified by linear address and limit.

```
 3                             1 1
 1                             6 5                             0
                                +-------------------------------+
                                |                               |
GDTR                            |            limit              | +0
                                |                               |
+-------------------------------+-------------------------------+
|                                                               |
|                             base                              | +2
|                                                               |
+---------------------------------------------------------------+


                                +-------------------------------+
                                |                               |
IDTR                            |            limit              | +0
                                |                               |
+-------------------------------+-------------------------------+
|                                                               |
|                             base                              | +2
|                                                               |
+---------------------------------------------------------------+



                                +-------------------------------+
                                |                               |
LDTR                            |            selector           |
                                |                               |
                                +-------------------------------+


                                +-------------------------------+
                                |                               |
TR                              |            selector           |
                                |                               |
                                +-------------------------------+
```

The GDTR holds a 16-bit limit and 32-bit linear base address of the GDT segment. The IDTR holds a 16-bit limit and 32-bit linear base address of the IDT segment.

The memory format for the pseudo-descriptors loaded by LIDT and LGDT, and stored by SIDT and SGDT are given in the following figure.

```
3                             1 1
1                             6 5                            0
+-----------------------------+------------------------------+
|                             |                              |
|        Base 15..0           |            limit             | +0
|                             |                              |
+-----------------------------+-------------+----------------+
                              |             |                |
                              |  00000000   |  Base 23..16   | +4
                              |             |                |
                              +-------------+----------------+
```

Pseudo-Descriptor format if operand size is 16

```
3                             1 1
1                             6 5                            0
+-----------------------------+------------------------------+
|                             |                              |
|        Base 15..0           |            limit             | +0
|                             |                              |
+-----------------------------+------------------------------+
                              |                              |
                              |         Base 31..16          | +4
                              |                              |
                              +------------------------------+
```

Pseudo-Descriptor format if operand size is 32

The LDTR holds a 16-bit selector for the LDT segment. This selector must name a segment descriptor located in the GDT, or a GP fault (int. 13) is raised. The TR holds the 16-bit selector for the TSS. The selector loaded into the TR can name a segment descriptor located in either an LDT or the GDT.

COMPATIBILITY NOTE:

On the 286 the GDTR and IDTR were only 40 bits long, but were loaded and stored from 48-bit memory operands. Loads were defined to ignore the upper 8 bits of the memory operand, and stores were defined to store an undefined value into the upper 8 bits of the memory operand. In order to maintain 286 compatibility, the operand length of the load instruction is used to determine whether to use the 286 load semantics (form the 32-bit base address by using only 24 bits from the memory operand, zero-extended to 32 bits), or to use the extended 386 semantics to get the full 32-bit base address from the memory operand. On stores, all 48 bits of the memory operand are always written out, with the full 32-bit base address written to the upper 32 bits of the memory operand. Note that on stores, the upper 8 bits may have different values on the 386 than on the 286. The value stored in the upper 8 bits was undefined in the 286 architecture specification. The value actually stored on the 286 has FF hex in the upper 8 bits of the memory operand. The value stored on the 386 when executing the same code will have 00 in the upper 8 bits.

## 2.3. Operand Addressing

### 2.3.1. Immediate Operands

The 8 and 16-bit immediate operand sizes supported by the 86 and 286 are supported, and are selected if the operand size of the instruction is 16. In most cases, a 16-bit immediate operand can be given as a sign-extended 8-bit immediate in order to conserve code space in the frequent case of a small number of significant digits in the immediate operand. An operand size of 32 generally indicates that 16 bit immediate fields are extended to 32 bits in the instruction, with the following three exceptions (also described in chapter 6). If a 32-bit operand size is specified, the ENTER and RET instructions take a 16-bit immediate zero-extended to 32-bits.

### 2.3.2. Register Operands

General register operands are specified by a three bit register number field in one of the following instruction formats:

```
 7       3 2   0
+---------+-----+
| opcode  | reg |
+---------+-----+

 7             0   15            8
+---------------+  +---+----+-----+
|    opcode     |  |mod| reg | R/M |
+---------------+  +---+----+-----+

 7             0   15            8
+---------------+  +---+----+-----+
|    opcode     |  |mod| opx | R/M |
+---------------+  +---+----+-----+

 7             0   15            8
+---------------+  +---+----+-----+
|    opcode     |  |1 1| reg | reg |
+---------------+  +---+----+-----+

 7             0   15            8
+---------------+  +---+----+-----+
|    opcode     |  |1 1| opx | reg |
+---------------+  +---+----+-----+
```

In the one-byte instruction format, the register number is given as the low-order 3 bits of the opcode byte. In the two-byte format, bits 11..13 give a register number (or an opcode extension). If the mod field in the MODRM byte is 11, then bits 8..10 also give a register number.

Register codes for all three formats are identical, and given by the following table. The register specified is dependent on the operand size given for the instruction (32, 16, or 8-bit).

| Reg Code | 32-bit Register | 16-bit Register | 8-bit Register |
|---|---|---|---|
| 000 | EAX | AX | AL |
| 001 | ECX | CX | CL |
| 010 | EDX | DX | DL |
| 011 | EBX | BX | BL |
| 100 | ESP | SP | AH |
| 101 | EBP | BP | CH |
| 110 | ESI | SI | DH |
| 111 | EDI | DI | BH |

### 2.3.3. Memory Operands

#### 2.3.3.1. Effective Address Computation

In order to enlarge the segment size beyond 64K, effective addresses can be 32-bits in addition to the current 16-bit size. The effective address is formed by adding together an optional base register, an optional index register, and an optional displacement. The 16-bit effective address modes permit an 8 or 16-bit displacement, and truncate the effective address at 16 bits (ignoring overflow). The 32-bit modes permit an 8 or 32-bit displacement, and produce a full 32-bit result. The 32-bit modes require that the base and index registers, if used, both hold valid 32-bit values. Since the 16-bit modes truncate the result at 16-bits, the base and index registers can have trash in the upper 16 bits.

In addition to providing the 32-bit analogs of the original 16-bit set of addressing modes, the 32-bit addressing modes have been expanded to permit any general register to be used as a base or index register, and to introduce scaled indexing.

#### 2.3.3.2. 16-Bit Addressing Modes

The original 16-bit 86/286 addressing modes are supported. That is, registers BX and BP are available as base registers, and SI and DI are available as index registers. The 16-bit memory addressing modes are given in the following table.

```
Segment  +   Base   +   Index   +          Displacement

  DS         ----       ----                   ----
  ES     +    BP    +    SI     +      8-bit Displacement
  CS          BX         DI           16-bit Displacement
  SS
Following choices added in 386:

  FS
  GS
```

Fig. 2.3.3.1: 16-bit Addressing Modes

The 16-bit addressing modes are encoded in the "MODRM" byte of an instruction, which follows the opcode byte. The MODRM byte is encoded as follows for the 16-bit addressing modes:

```
 7 6 5   3 2   0'
+---+-----+-----+
|mod| xxx | R/M |
+---------+-----+
```

The R/M field encodes the base and/or index register to be used in forming the memory address, as given in the following table:

| R/M | Registers |
|---|---|
| 000 | [BX+SI] |
| 001 | [BX+DI] |
| 010 | [BP+SI] |
| 011 | [BP+DI] |
| 100 | [SI] |
| 101 | [DI] |
| 110 | [BP]*** |
| 111 | [BX] |

The mod field is encoded as follows:

| mod | Displacement Size |
|---|---|
| 00 | no Displacement*** |
| 01 | 8-bit Displacement |
| 10 | 16-bit Displacement |
| 11 | indicates Register Operand, see 2.3.2 |

*** mod=00 and base=110 (BP register) indicate a 16-bit displacement with no base register.

As in the 286, the DS register is the default segment base for memory references, unless BP is used as the base register, in which case SS is the default segment base. If a different segment base is desired, a segment override prefix can be used to select a different segment.

### 2.3.3.3. 32-Bit Addressing Modes

The 32-bit addressing modes are extended to allow any register to be used as a base, any register to be used as an index, and to provide index scaling by 1, 2, 4, or 8. The double indexed forms (EBX+ESI, EBX+EDI, etc.), and the scaled index forms, require an extra byte following the MODRM byte, immediately preceding any displacement bytes.

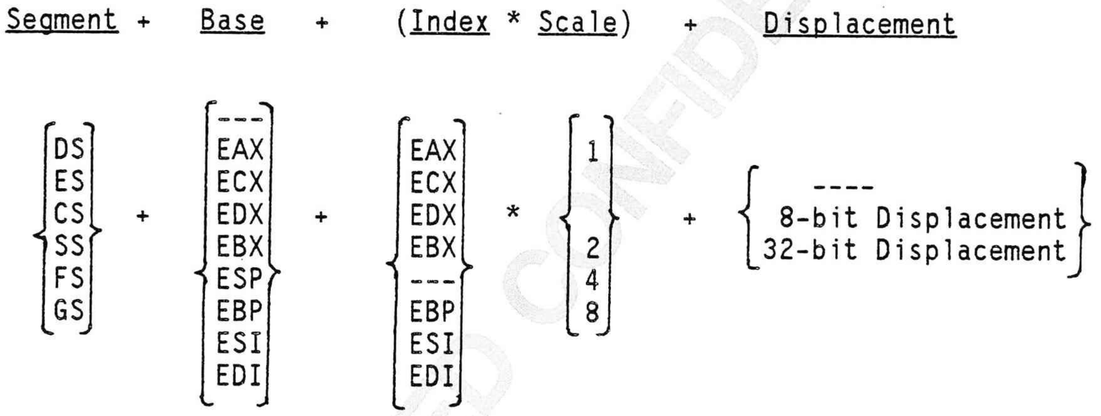
Segment + Base + (Index * Scale) + Displacement
DS
ES
CS
SS
FS
GS
+
---
EAX
ECX
EDX
EBX
ESP
EBP
ESI
EDI
+
EAX
ECX
EDX
EBX
---
EBP
ESI
EDI
*
1
2
4
8
+
----
8-bit Displacement
32-bit Displacement

Fig. 2.3.3.2: 32-bit Addressing Modes

Using the same code used for standard register access, and using the code for ESP (100) as the 2-byte escape, gives the following addressing mode encoding:

```
 7 6 5   3 2   0
+---+-----+-----+
|mod| xxx | base|          mod <> 11, base <> 100
+---------+-----+


 7 6 5   3 2   0      7 6 5   3 2   0
+---+-----+-----+    +---+-----+-----+
|mod| xxx | 100 |    | ss|index| base|   mod <> 11
+---------------+    +---+-----+-----+
```

In the one-byte form, ss is taken as 00, and index as 100. This provides no scaling, and no index component.

The base and index fields encode 32-bit registers using the codes given above for register operands, with the following two exceptions. An index code of 100 (ESP) indicates no index component, in which case the SS field is ignored. A base code of 101 (EBP) and mod=00 indicate no base register, with a 32-bit displacement.

The address modes are encoded with mod and base fields as given in the following table. The "Address Mode" field in the table gives the default segment register, along with the base register (shown inside square brackets), and the displacement, if any, indicated as Disp8 for an 8-bit displacement, or Disp32 for a 32-bit displacement. The default segment register is used unless the instruction is preceded by a segment override prefix.

| mod | base | Address Mode |
|---|---|---|
| 00 | 000 | DS:[EAX] |
| | 001 | DS:[ECX] |
| | 010 | DS:[EDX] |
| | 011 | DS:[EBX] |
| | 100 | SS:[ESP] |
| | 101 | DS:Disp32 |
| | 110 | DS:[ESI] |
| | 111 | DS:[EDI] |
| 01 | 000 | DS:Disp8[EAX] |
| | 001 | DS:Disp8[ECX] |
| | 010 | DS:Disp8[EDX] |
| | 011 | DS:Disp8[EBX] |
| | 100 | SS:Disp8[ESP] |
| | 101 | SS:Disp8[EBP] |
| | 110 | DS:Disp8[ESI] |
| | 111 | DS:Disp8[EDI] |
| 10 | 000 | DS:Disp32[EAX] |
| | 001 | DS:Disp32[ECX] |
| | 010 | DS:Disp32[EDX] |
| | 011 | DS:Disp32[EBX] |
| | 100 | SS:Disp32[ESP] |
| | 101 | SS:Disp32[EBP] |
| | 110 | DS:Disp32[ESI] |
| | 111 | DS:Disp32[EDI] |

| index | Address Mode |
|---|---|
| 000 | [EAX*ss] |
| 001 | [ECX*ss] |
| 010 | [EDX*ss] |
| 011 | [EBX*ss] |
| 100 | no index |
| 101 | [EBP*ss] |
| 110 | [ESI*ss] |
| 111 | [EDI*ss] |

The ss field is encoded as follows:

| ss | Scale Factor |
|---|---|
| 00 | 1 |
| 01 | 2 |
| 10 | 4 |
| 11 | 8 |

As in the 286, the DS register is the default segment base for memory references, unless EBP or ESP is used as the base register, in which cases SS is the default segment base. If a different segment base is desired, a segment override prefix can be used to select a different segment.

Example: By specifying a scaled index in combination with a displacement-only addressing mode, the important case of a single scaled index into a STATIC array can be handled.

MOV EDX, TABLE[ESI*4] would be encoded as:

```
+---------+  +--+---+---+  +--+---+---+  +--------------+
| opcode  |  |00|010|100|  |10|110|101|  | 32-bit displ.|
+---------+  +--+---+---+  +--+---+---+  +--------------+
```

Example: Stack relative addressing is done with the two byte form by using ESP as a base register with the "null" index code (100), and a choice of 0 (stack top), 8, or 32-bit (inner stack element references) displacement.

```
MOV  EDX, 0[ESP]  ; would be encoded as:

+--------+  +--+---+---+  +--+---+---+
| opcode |  |00|010|100|  |00|100|100|
+--------+  +--+---+---+  +--+---+---+


MOV  EDX, 24[ESP] ; would be encoded as:
+--------+  +--+---+---+  +--+---+---+  +----------+        |
| opcode |  |01|010|100|  |00|100|100|  | 00011000 |        |
+--------+  +--+---+---+  +--+---+---+  +----------+        |
```

Example: An address mode with no index component (e.g. Base+Displacement) can be specified in either of two ways. If the base register is any register except ESP, the one-byte MODRM form can be used. Or the 2-byte MODRM form can be used with index=100 (null index) and ss=00.

```
MOV  EDX, 0[EAX]  ; could be encoded as:

+--------+  +--+---+---+
| opcode |  |00|010|000|
+--------+  +--+---+---+

; or could use a 2-byte MODRM form...

+--------+  +--+---+---+  +--+---+---+
| opcode |  |00|010|100|  |00|100|000|
+--------+  +--+---+---+  +--+---+---+


MOV  EDX, DISPONLY   ; could be encoded as:

+--------+  +--+---+---+  +---------------------+
| opcode |  |00|010|101|  | 32-bit displacement |
+--------+  +--+---+---+  +---------------------+

; or could use a 2-byte MODRM form...

+--------+  +--+---+---+  +--+---+---+  +--------------+
| opcode |  |00|010|100|  |00|100|101|  | 32-bit displ.|
+--------+  +--+---+---+  +--+---+---+  +--------------+
```

Example: The EBP register can be used as an index register with a 32-bit address mode. In this case, the default segment register is DS, not SS. SS is the default segment register only if EBP or ESP is used as the base register. The 386 differentiates the use of EBP as a base or index register. This distinction is clear in the hardware, since the base and index fields are distinct. Assembler languages need to clearly identify when EBP is used as a base vs. as an index in cases where two unscaled registers are given in an address mode.

```
MOV  EDX, 100[EBP]; EBP is a base, segment is SS:

+---------+  +--+---+---+  +---------+
| opcode |   |01|010|101|  |01100100|
+---------+  +--+---+---+  +---------+

MOV  EDX, Table[EBP*4] ; EBP is an index, segment is DS:

+---------+  +--+---+---+  +--+---+---+  +---------------+
| opcode |   |00|010|100|  |10|101|101|  | 32-bit displ.|
+---------+  +--+---+---+  +--+---+---+  +---------------+


MOV  EBX, [EBP+ESI] ; Assume EBP is base, ESI index.
                    ; Segment then is SS:

+---------+  +--+---+---+  +--+---+---+  +---------+
| opcode |   |01|011|100|  |00|110|101|  |00000000|
+---------+  +--+---+---+  +--+---+---+  +---------+

MOV  EBX, [ESI+EBP] ; Assume ESI is base, EBP index.
                    ; Segment then is DS:

+---------+  +--+---+---+  +--+---+---+
| opcode |   |00|011|100|  |00|101|110|
+---------+  +--+---+---+  +--+---+---+
```

## 2.4. Data Alignment

There are no alignment requirements on application level data types. Integers, Ordinals, Reals, Decimals, and Pointers can be located at any address. The string types (BCD, Byte, and Bit) are byte granular and can start at any byte address.

If paging is enabled, alignment of certain system objects is required to ensure that page faults do not occur when accessing these objects except at the first reference. TSSs must be aligned or wired into physical memory so that no page faults can occur in the middle of a task switch operation. A page fault can be accepted if it occurs on the first reference to the TSS. The OS should take care to align TSSs to 128 byte boundaries if they are to be paged, to ensure that they do not cross page boundaries.

Page tables are aligned at page boundaries, for efficient access, and to permit the page tables to be easily paged.

Although there are no requirements on alignment other than those stated above, unaligned data may require additional bus cycles at each access, depending on the characteristics of the bus interface. It is reccommended that all of the basic data types be aligned to an address that is a multiple of the length of the data type. Pointers and TEMPREAL data should be aligned to a 4-byte boundary.

# 3. PROTECTED MODE MEMORY ADDRESSING

## 3.1. Linear Address Formation

In protected addressing (PE bit in MSW is 1, and not in Virtual 8086 mode), the linear address is formed by adding together the 16 or 32-bit effective address with a segment base value obtained from the descriptor table entry selected by the index in the segment register given (or assumed) for the instruction. The linear address of a memory reference to a segment is formed by adding the base address in the descriptor to the effective address computed in an instruction, and checking the effective address against the segment limit.

The effective address size used for implicit code references (instruction prefetch) is always 32 bits. That is, the full 32-bit EIP register is used for implicit code references. Compatibility with the 16-bit IP on the 286 is enforced by segment limit checks, and by the availability of a full complement of control transfer instructions which operate on the 16-bit IP register. 286 code will have segment limits smaller than 64K, so that limit violations will occur if IP is incremented beyond 64K (instruction fetch off end of a 64K segment).

The effective address size used for implicit stack addressing (e.g. PUSH, POP, CALL, RET, etc.) is controlled by the B bit in the segment descriptor loaded into the SS register. B=0 (286 compatible setting) indicates that a 16-bit effective address (using the 16-bit SP register) is to be used for implicit stack addressing. B=1 indicates that 32-bit effective addresses are to be used.

COMPATIBILITY NOTE:

> The handling of stack overflow and underflow is exactly the same as the 286. A 286 program with a 64K stack would wrap around, with no fault raised, if stack overflow or underflow occurs. A limit violation is raised on the 286 only if a PUSH is attempted with SP=1, or a POP with SP=0FFFF. Also, on the 286, SP is usually initialized to 0, so that the first PUSH will "wrap-around" to use the top word on the stack. It is important that the 386 not fault in this case. The B bit in the segment discriptor allows this kind of wraparound to occur.

## 3.2. SELECTOR FORMAT

386 Selectors have the same format as on the 286. Selectors are 16 bits in size. The descriptor index is in bits 15..3, the Table Index is in bit 2, and the Requested Priority Level is in bits 1..0.

```
 15                               3 2 1 0
+---------------------------------+
|                          |T|     |
|     DESCRIPTOR INDEX     | |RPL|
|                          |I|     |
+---------------------------------+
```

Selector Format

The TI bit indicates whether the descriptor for the segment named by the selector is in the GDT (TI=0), or in the LDT (TI=1). The descriptor index field is used as an index into the given table to obtain the descriptor for the segment named by the selector.

The RPL field can be used to "weaken" the privilege of this selector. When a selector is tested for accessibility, the DPL (Descriptor Privilege Level, stored in the descriptor) is compared against the larger of RPL and CPL (Current Privilege Level). If DPL is less than MAX(RPL, CPL), access to the segment named by the selector is denied. RPL=0 effectively disables the RPL field, causing DPL to be compared directly to CPL.

## 3.3. DESCRIPTOR FORMATS

The descriptors available within the 386 protection system include all of the 286 descriptor types. The 286 descriptors have been extended in two ways for the 386: unassigned descriptor TYPE numbers have been assigned to new 386 descriptors, and reserved fields in 286 descriptor types have been assigned meanings. In order for a 286 system to run on the 386, none of the unassigned TYPE codes can be used, and the reserved fields in the 286 descriptors must be zero. This requires 286 operating systems to initialize these fields to zero even though the 286 processor will not access them.

COMPATIBILITY NOTE:

> The use of the "reserved" bits in the descriptors requires that 286 operating systems set these fields to zeros. Use of codes left "Undefined" in the 286 for new 386 descriptor types requires that 286 operating systems use 0 or 80 as undefined types.

All of the protection and level concepts are identical to the iAPX 286. The 386 extensions are limited to providing larger segment base and limit values for memory segments, two new segment attributes, providing 386 Task State Segments for the expanded 386 machine state, and new CALL, TRAP, and INTERRUPT gates for 32-bit operation.

The extensions to the descriptors are described in three subsections: memory segments, special memory segments, and gates.

### 3.3.1. Memory Segments

As on the 286, two types of memory segments are provided: data segments and code segments. The two memory segment types are very similar, differing only in the access rights provided.

Both code and data segment descriptors have been extended from the 286 formats by using the reserved field to expand the base and limit fields, and to define new segment attributes. An 8 byte descriptor format provides a 32 bit base, and a 20 bit limit (max. segment size of 1 megabyte) with 1 byte granularity, or a 32 bit limit with page (4K) granularity. Without using the fractured extensions of the descriptors, these descriptor formats provide a 24-bit base, and a 16 bit limit (64K size) with 1 byte granularity, or a 28-bit limit (256 Megabytes) with page granularity.

### 3.3.1.1. Data Segments

```
 31          24              16            8 7 6 5 4 3 2 1 0
+-------------+---------------+-------------+---------------+
|                             |                             |
|   Segment Base  15..0       |   Segment Limit 15..0       | +0
|                             |                             |
+-------------+-+-+-+-+-------+-+---+-+-+-+-+-+-------------+
|             | | | |A| Limit | |   | | | | | |             |
| Base 31..24 |G|B|0|V|       |P|DPL|1|0|E|W|A| Base 23..16 | +4
|             | | | |L| 19..16| |   | | | | | |             |
+-------------+-+-+-+-+-------+-+---+-+-+-+-+-+-------------+
```

Memory DATA Segment Descriptor

The G bit in the descriptor is used to indicate the granularity of the limit field. It has no effect on the Base field, since the Base address is given as a full 32-bit "byte granular" address. If G is zero (286 compatible), the Limit field is byte granular, and is extended to 32 bits with high-order zeros.

If the G bit is 1, the Limit field has page (4K) granularity, and is shifted left 12 bits (with low order <u>ones</u> inserted) when loaded.

Example:
If G=1, E=0, Base=1000 hex, and Limit=0, the descriptor defines a segment with base address 1000 hex (4096 decimal), and limit of 0FFF hex (4095 decimal). This defines a segment that is one page long, encompassing the second page frame in the linear address space.

Example:
If G=1, E=0, Base=0, and Limit=000FFFFF, the descriptor defines a segment with base address 0, and limit of FFFFFFFF. This defines a segment that encompasses the entire 32-bit linear address space.

If the E bit is 1, indicating an expand-down segment, the limit field indicates the lower limit of the segment. The limit field indicates an address that is one less than the lowest legal address. The segmentation hardware will trap if an effective address smaller than or equal to the limit value is used in forming an address. The upper limit of the segment is given by the B bit. B=0 indicates a 64K upper limit (286 compatible), and B=1 indicates a 4G upper limit.

PROGRAMMING NOTE:
Expand down segments which have B=1 imply a 4G upper limit

for the segment. The Limit field in expand down segments gives the lower limit of the segment. The upper bits of the limit field are zero if G=0 (byte granular limit), so that the largest byte-granular limit is 1 Meg. This means that big (B=1) expand down (E=1) segments must always use a page granular (G=1) limit in order to specify a lower limit that is close to the implied 4G upper limit. Byte granular limits make no sense for these big expand down segments, since they imply a very large segment, encompassing almost the entire linear address space (lower limit in the range 0..1Meg, upper limit 4G). A byte-granular limit for these segments implies a lower limit for the segment that is less than 1 Megabyte, which combined with an upper limit of 4G means a very large segment!

This means that the smallest expand-down segment with a 4G upper bound is one page (4K), with expansion supported in page increments. If this is too large, and there is no need to expand beyond 64K bytes, the expand-down segments with 64K upper bound can be used.

The B bit is examined in descriptors loaded into the SS register to determine the "size" of the stack pointer register (16-bit SP or 32-bit ESP) to be used in implicit stack references in PUSH, POP, CALL, etc. instructions. For expand-down segments, the B bit is also used to give the implied upper limit, whether or not the descriptor is loaded into the SS register.

B=0 (286 compatible setting) in a descriptor loaded into the SS register indicates that implicit stack pointer references (for PUSH, POP, CALL, etc.) should use the 16-bit SP register. B=0 also indicates an upper limit of 64K for expand-down segments. If B=0 and E=1, the segmentation hardware will trap if an effective address larger than 64K is used in forming an address using this descriptor.

B=1 in a descriptor loaded into the SS register indicates that implicit stack pointer references should use the 32-bit ESP register as the stack pointer. B=1 also indicates an upper limit of 4G for expand-down segments.

Bit 52 (marked AVL) is available for software use. It will not be defined in any future processor. Bit 53 (marked 0), must be zero for compatibility with future processors.

PROGRAMMING NOTE:

The following code sequence can be used to "pack" a 32-bit base, 20-bit limit, and 16-bit attribute field into the 64-bit descriptor format. Assume the attribute field is stored in the same format as in the descriptor, with 0's in the 4 limit bits.

```
MOV  BX, ATTR16    ; Load Attribute WORD
SHL EBX, 16        ; Chinese puzzle rotate
MOV EDX, Base      ; Load Base Address
ROL EDX, 16        ; Put high bytes into byte regs.
MOV  BL, DH        ; Base 31..24
MOV  BH, DL        ; Base 23..16
ROR EBX, 8         ; Rotate to final alignment
MOV EAX, Limit     ; Load Limit
MOV  DX, AX        ; Limit 15..0 with Base 15..0
AND EAX, 000F0000H; Mask Limit 19..16
OR  EBX, EAX       ; OR into high order word.
MOV Descr, EDX     ; Store low word
MOV Descr+4, EAX   ; Store high word
```

This sequence requires 19 clocks overhead, in addition to 19 clocks (0 wait state pipelined memory) required to load and store registers, which will be needed in any case.

The sequence to "unpack" the segment base is:

```
MOV EAX, Descr       ; Load low word
MOV EDX, Descr+4     ;      high word
SHRD EAX, EDX, 16    ; Align Base 23..0
AND  EAX, 00FFFFFFH  ; Clear AR byte
AND  EDX, FF000000H  ; Mask Base 31..24
OR   EAX, EDX        ; EAX holds base.
```

PROGRAMMING NOTE:

If B=0, E=1, and the segment limit is larger than 64K, all effective addresses within the segment will cause a protection trap, since they will either be larger than 64K (a violation of the 64K upper limit indicated by B=0 and E=1), or will be less than the lower limit given in the descriptor.

PROGRAMMING NOTE:

If a 16-bit software system is modified to use a stack larger than 64K, all addresses which reference the stack must be changed to use 32-bit effective addresses, in order to cover the span of the larger stack segment. The expand down segment attributes can be used to check that no 16-bit effective addresses are used to address within the stack segment by defining the stack segment as a Big Expand-down segment (i.e. B=1 and E=1) with a limit larger than 64K. This will raise segment limit faults if any 16-bit effective addressing is done within the stack segment, since the 16-bit effective address must be smaller than 64K, which is smaller than the lower limit for the expand-down segment.

### 3.3.1.2. Code Segments

```
 31             24              16              8 7 6 5 4 3 2 1 0
+---------------+---------------+---------------+---------------+
|                               |                               |
|   Segment Base  15..0         |     Segment Limit 15..0       | +0
|                               |                               |
+---------------+-+-+-+-+-------+-+---+-+-+-+-+-+---------------+
|               | | | |A| Limit | |   | | | | | |               |
|  Base 31..24  |G|D|0|V|       |P|DPL|1|1|C|R|A|  Base 23..16  | +4
|               | | | |L| 19..16| |   | | | | | |               |
+---------------+-+-+-+-+-------+-+---+-+-+-+-+-+---------------+
```

Memory CODE Segment Descriptor

The D bit indicates the default for operand lengths and effective addresses when the processor is executing code contained in this segment, as descibed in the OPERATIONS section. If 0, 16 bit defaults are assumed, as in the 286. If 1, 32-bit operands and effective addresses are assumed.

### 3.3.2. Special Memory Segments

Descriptor entries can also point to Task State Segments or Descriptor Tables, which are segments with special significance to the processor.

```
 31            24               16               8 7 6 5 4 3 2 1 0
+---------------+---------------+---------------+---------------+
|                               |                               |
|    Segment Base  15..0        |     Segment Limit 15..0       | +0
|                               |                               |
+---------------+-+-+-+-+-------+-+---+-+-------+---------------+
|               | | | |A| Limit | |   | |       |               |
|  Base 31..24  |G|0|0|V|       |P|DPL|0| TYPE  |  Base 23..16  | +4
|               | | | |L| 19..16| |   | |       |               |
+---------------+-+-+-+-+-------+-+---+-+-------+---------------+
```

Special Memory Segment Descriptor

The TYPE field (bits 11..8 of the high order DWORD) identifies the descriptor as one of several possible types. Note that the descriptor format is identical to Data segment descriptors except that no "accessed" bit is provided, and no "B" bit is provided.

The TYPE field is encoded as follows, for Special Segments and Gates:

| TYPE | Defines |
|---|---|
| 0 | Undefined |
| 1 | Available 286 TSS |
| 2 | LDT |
| 3 | Busy 286 TSS |
| 4 | Call Gate |
| 5 | Task Gate |
| 6 | 286 Interrupt Gate |
| 7 | 286 Trap Gate |
| 8 | Undefined |
| 9 | Available 386 TSS |
| A | Undefined |
| B | Busy 386 TSS |
| C | 386 Call Gate |
| D | Undefined |
| E | 386 Interrupt Gate |
| F | 386 Trap Gate |

### 3.3.2.1. TYPE = 2, Local Descriptor Table

A Local Descriptor Table segment contains descriptors. The LDT is the only decriptor table described by a segment descriptor. The GDT and IDT are described by explicitly loading the base and limit values into special processor registers.

An LDT segment descriptor can only appear in the GDT. An attempt to identify an LDT with a selector which has TI=1 (indicating the selector names a segment in the LDT) will result in a protection fault.

PROGRAMMING NOTE:

> When a descriptor is read from the LDT, all 8 bytes must be within the LDT limit value, or a limit violation occurs. In order to avoid a limit fault on reading descriptor n, the limit must be larger than or equal to 8n-1.

### 3.3.2.2. TYPE = 1, 3 286 Task State Segment

A Task State Segment (TSS) is a special fixed format segment which contains all the state information for a task, and a linkage field to permit nesting of tasks. The TYPE field is used to indicate whether the task is currently active (or on a chain of active tasks) or available. TYPE 1 is an available TSS, type 3 is a busy TSS.

A TSS descriptor can only appear in the GDT. An attempt to identify a TSS with a selector which has TI=1 (indicating the selector names a segment in the LDT) will result in a protection fault.

286 TSS Format

The physical format of a 286 Task State Segment is given below. This defines the first 44 bytes of the segment. The remainder of the segment can be defined by software convention. The segment can have any limit greater than or equal to 43. If the limit value is less than this lower limit, an invalid TSS fault occurs if a task switch operation is attempted through the TSS.

A system error is permitted to occur (implementation dependent) if a page fault occurs in the middle of accessing a TSS, so the OS must ensure that either the TSS does not span a page boundary, or the entire TSS is resident before dispatching a task.

```
TYPE TSS = RECORD

          LINK : Selector; (* Selector for previous TSS *)

     (* Static Portion *)

          SP0  : WORD;  (* Level 0 stack pointer *)
          SS0  : WORD;
          SP1  : WORD;  (* Level 1 stack pointer *)
          SS1  : WORD;
          SP2  : WORD;  (* Level 2 stack pointer *)
          SS2  : WORD;

     (* Register Images *)
          IP : WORD;
          Flags : WORD;
          AX, CX, DX, BX, SP, BP, SI, DI : WORD;
          ES, CS, SS, DS : WORD;  (* Selector registers *)

          LDT  : Selector;
```

Given as a memory image, the 286 TSS looks like the following:

### 286 TSS Format

```
 1
 6                              0
+--------------------------------+
|             LINK               | 0
+--------------------------------+
|              SP0               | 2
+--------------------------------+
|              SS0               | 4
+--------------------------------+
|              SP1               | 6
+--------------------------------+
|              SS1               | 8
+--------------------------------+
|              SP2               | A
+--------------------------------+
|              SS2               | C
+--------------------------------+
|               IP               | E
+--------------------------------+
|             FLAGS              | 10
+--------------------------------+
|               AX               | 12
+--------------------------------+
|               CX               | 14
+--------------------------------+
|               DX               | 16
+--------------------------------+
|               BX               | 18
+--------------------------------+
|               SP               | 1A
+--------------------------------+
|               BP               | 1C
+--------------------------------+
|               SI               | 1E
+--------------------------------+
|               DI               | 20
+--------------------------------+
|               ES               | 22
+--------------------------------+
|               CS               | 24
+--------------------------------+
|               SS               | 26
+--------------------------------+
|               DS               | 28
+--------------------------------+
|              LDT               | 2A
+--------------------------------+
```

### 3.3.2.3. TYPE = 9, 11 386 Task State Segment

This descriptor is similar to types 1 and 3, except the segment they point to has a different, expanded format given below. The TYPE field is used to indicate whether the task is currently active (or on a chain of active tasks) or available. TYPE 9 is an available TSS, type 11 is a busy TSS.

386 TSS Format

The physical format of a 386 Task State Segment is given below. This defines the first 104 bytes of the segment. The remainder of the segment can be defined by software convention. The segment can have any limit greater than or equal to 86. If the limit value is less than this lower limit, an invalid TSS fault occurs if a task switch operation is attempted through the TSS.

A system error is permitted to occur (implementation dependent) if a page fault occurs in the middle of accessing a TSS, so the OS must ensure that either the TSS does not span a page boundary, or the entire TSS is resident before dispatching a task.

```
TYPE TSS = RECORD

         LINK : DWORD;    (* LINK to previous TSS (if any) *)

     (* Static Portion *)

         ESP0 : DWORD;  (* Level 0 stack pointer *)
         SS0  : DWORD;
         ESP1 : DWORD;  (* Level 1 stack pointer *)
         SS1  : DWORD;
         ESP2 : DWORD;  (* Level 2 stack pointer *)
         SS2  : DWORD;

     (* Register Images *)

         CR3 : DWORD; (* Swapped on a task switch *)
         EIP : DWORD;
         EFlags : DWORD;
         EAX, ECX, EDX, EBX, ESP, EBP, ESI, EDI : DWORD;
         ES, CS, SS, DS, FS, GS : DWORD;
         LDT  : DWORD;
         TASK ATTRIBUTES: DWORD; (* low order bit is DEBUGTRAP.
                If DEBUGTRAP is 1 in the "new" TSS on a Task
                Switch, raise a Debug Exception *)

END
```

Given as a memory image, the 386 TSS looks like the following:

386 TSS Format

```
 31                                                            0
+--------------------------------+-------------------------------+
|0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0|             LINK              | 0
+--------------------------------+-------------------------------+
|                              ESP0                              | 4
+--------------------------------+-------------------------------+
|0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0|              SS0              | 8
+--------------------------------+-------------------------------+
|                              ESP1                              | C
+--------------------------------+-------------------------------+
|0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0|              SS1              | 10
+--------------------------------+-------------------------------+
|                              ESP2                              | 14
+--------------------------------+-------------------------------+
|0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0|              SS2              | 18
+--------------------------------+-------------------------------+
|                               CR3                              | 1C
+----------------------------------------------------------------+
|                               EIP                              | 20
+----------------------------------------------------------------+
|                             EFLAGS                             | 24
+----------------------------------------------------------------+
|                               EAX                              | 28
+----------------------------------------------------------------+
|                               ECX                              | 2C
+----------------------------------------------------------------+
|                               EDX                              | 30
+----------------------------------------------------------------+
|                               EBX                              | 34
+----------------------------------------------------------------+
|                               ESP                              | 38
+----------------------------------------------------------------+
|                               EBP                              | 3C
+----------------------------------------------------------------+
|                               ESI                              | 40
+----------------------------------------------------------------+
|                               EDI                              | 44
+----------------------------------------------------------------+
```

386 TSS Format (cont.)

```
 31                                                             0
+-------------------------------+-------------------------------+
|0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0|              ES               | 48
+-------------------------------+-------------------------------+
|0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0|              CS               | 4C
+-------------------------------+-------------------------------+
|0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0|              SS               | 50
+-------------------------------+-------------------------------+
|0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0|              DS               | 54
+-------------------------------+-------------------------------+
|0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0|              FS               | 58
+-------------------------------+-------------------------------+
|0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0|              GS               | 5C
+-------------------------------+-------------------------------+
|0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0|             LDT               | 60
+-------------------------------+-----------------------------+-+
|                     A V A I L A B L E                       |T| 64
+-------------------------------+-----------------------------+-+
```

### 3.3.3. Gates

Gate descriptors provide protected, indirect CALLs. Gates support protected inter-level calls, and inter-level interrupts. The 386 defines Call gates, Trap gates, Interrupt gates, and Task gates. Two classes of gates are defined: 286 gates, with the original 286 TYPE assignments, format and semantics, and 386 gates, with new types (unassigned in the 286), expanded format (using the 286 reserved word), and different semantics.

```
 31           24             16            9 8 7 6 5 4 3 2 1 0
+--------------+--------------+--------------+--------------+
|                             |                             |
|        Selector             |        Offset 15..0         |
|                             |                             |
+-----------------------------+-+---+-+------+----+---------+
|                             | |   | |      |    |         |
|        Offset 31..16        |P|DPL|0| TYPE |0 0 0|Wrd Count|
|                             | |   | |      |    |         |
+-----------------------------+-+---+-+------+----+---------+
```

GATE Descriptor

COMPATIBILITY NOTE:

> The "reserved" word on the 286 must be set to zero by 286 operating systems in order to ensure portability to the 386.

#### 3.3.3.1. TYPE = 4, 286 Call Gate

A Call Gate indicates that control is to transfer to a procedure defined within the currently executing task. The virtual address gives the starting address of the procedure to invoke. The Word Count field gives the number of parameter words (16-bit words) to copy to the called procedure. These parameter words are copied from the old stack to the new stack before the return address is pushed and the new procedure is entered.

Calls through this gate use exactly the 286 semantics for call gates, pushing a 16-bit IP and a 16-bit SP value. Calls to the same level push CS, then IP, both as 16-bit values. Calls to inner levels push SS, SP, CS, then IP, all as 16-bit values.

PROGRAMMING NOTE:

> Return through this gate should be with a 286 FAR return (16-bit) so that upon return the proper information is popped off the stack.

#### 3.3.3.2. TYPE = 5, Task Gate

A Task gate indicates that a process switch mechanism is to be used to transfer control. The current machine state is saved in the current task's TSS, then the machine state is reloaded from the TSS pointed to by the virtual address in this gate. The selector portion of the virtual address must identify the TSS for the new task. The offset portion of the virtual address is ignored.

#### 3.3.3.3. TYPE = 6, 286 Interrupt Gate

An Interrupt Gate is similar to a Call Gate and a Trap Gate. It specifies a procedure that enters with interrupts disabled and returns via an interrupt return instruction (IRET). The Word Count field is ignored. Traps through this gate use exactly the 286 semantics for Interrupt gates, pushing a 16-bit IP and a 16-bit SP value. Interrupts to the same level push the 16-bit Flags, a 16-bit CS, then a 16-bit IP. Interrupts to inner levels push SS, SP, Flags, CS, then IP, all as 16-bit values.

PROGRAMMING NOTE:

> Return through this gate should be with a 286 (16-bit) IRET instruction so that upon return the proper information is popped off the stack.

#### 3.3.3.4. TYPE = 7, 286 Trap Gate

A Trap Gate is similar to a Call Gate and an Interrupt Gate. It specifies a procedure that enters with interrupt status unchanged and returns via an interrupt return instruction (IRET). The Word Count field is ignored.

Traps through this gate use exactly the 286 semantics for Trap gates, pushing a 16-bit IP and a 16-bit SP value. Interrupts to the same level push the 16-bit Flags, a 16-bit CS, then a 16-bit IP. Interrupts to inner levels push SS, SP, Flags, CS, then IP, all as 16-bit values.

PROGRAMMING NOTE:

> Return through this gate should be with a 286 (16-bit) IRET instruction so that upon return the proper information is popped off the stack.

#### 3.3.3.5. TYPE = 12, 386 Call Gate

A Call Gate indicates that control is to transfer to a procedure defined within the currently executing task. The virtual address gives the starting address of the procedure to invoke. The Word Count field gives the number of parameter dwords (32-bit dwords) passed to the called procedure. On inter-level CALLs, these parameter dwords are copied from the old stack to the new stack after the old stack pointer is pushed, and before the return address is pushed and the new procedure is entered.

Calls through this gate extend the 286 semantics for call gates by pushing 32-bit EIP and ESP (if needed) images, and by pushing the CS and SS (if needed) selectors as 32 bit quantities with the selectors in the low order bits, and the upper 16 bits undefined.

PROGRAMMING NOTE:

> Return from the procedure called by this gate should be with a FAR return with a 32-bit operand size, so that upon return the proper information is popped off the stack.

### 3.3.3.6. TYPE = 14, 386 Interrupt Gate

An Interrupt Gate is similar to a Call Gate and a Trap Gate. It specifies a procedure that enters with interrupts disabled and returns via an interrupt return instruction (IRET). The Word Count field is ignored.

Transitions through this gate extend the 286 semantics by pushing 32-bit EIP, EFLAGS, and ESP (if needed) images, and by pushing the CS and SS (if needed) selectors as 32 bit quantities with the selectors in the low order bits, and the upper 16 bits undefined.

PROGRAMMING NOTE:

> Return from the procedure called by this gate should be with an IRET with a 32-bit operand size, so that upon return the proper information is popped off the stack.

### 3.3.3.7. TYPE = 15, 386 Trap Gate

A Trap Gate is similar to a Call Gate and an Interrupt Gate. It specifies a procedure that enters with interrupt status unchanged and returns via an interrupt return instruction (IRET). The Word Count field is ignored.

Transitions through this gate extend the 286 semantics by pushing 32-bit EIP, EFLAGS, and ESP (if needed) images, and by pushing the CS and SS (if needed) selectors as 32 bit quantities with the selectors in the low order bits, and the upper 16 bits undefined.

PROGRAMMING NOTE:

> Return from the procedure called by this gate should be with an IRET with a 32-bit operand size, so that upon return the proper information is popped off the stack.

PROTECTED CONFIDENTIAL

## 3.4. PAGING

The paging mechanism is positioned beneath the segmentation mechanism of the 386. That is, during physical address generation a virtual address is first relocated by the segmentation mechanism to a "linear address" by adding the segment base address corresponding to the selector portion of the address to the offset part of the virtual address. This linear address is then translated by a page table to obtain the actual physical address. If paging is not in use, the "linear" address is taken as the physical address. The relocation performed by the segmentation mechanism is described in previous sections.

The paging mechanism introduces a third address space to a 386 system: a "linear" address space, which intervenes between the segmented (but not paged) virtual address space seen by the programmer, and the physical address space used on the processor bus.

A limited amount of protection information is included in the paging mechanism, in order to facilitate sharing of second level page tables, and to allow large segments in protected mode to have some sub-structure. This is necessary to support UNIX in the 386 FLAT model of computation. A limited amount of protection information at the page level also supports protection in virtual 8086 mode.

### 3.4.1. Linear Address Space Partitioning

```
31                22 21               12 11                     0
+-------------------+-------------------+-----------------------+
|                   |                   |                       |
|     Directory     |       Page        |        OFFSET         |
|                   |                   |                       |
+-------------------+-------------------+-----------------------+
```

Figure 3.3.1: Linear Address

The bits of a "linear" address are partitioned as shown to implement two-level paging of the linear address space with a 4K page size. The offset field is 12 bits, to accomodate a 4K page size. In order to be able to page the page table structure, and to support efficient processor access to the page table structure, the second level tables (Page Tables) are contained in page frames, and are page aligned. Since the entries in the Page table are 4 bytes wide, there are 1K entries in each table, so the Page fields in the linear address are 10 bits wide.

Page directories are located on page boundaries, and are at most one page in length. This simplifies access to the memory resident page tables, since no adds need be done to index into the directory. The directory index (bits 31..22 of the linear address) are shifted right 20 and concatenated to the base value in CR3 to index into the first level page table. This permits efficient access to the page table structure, and permits simple processors (such as the numerics processor and DMA processors) to efficiently access the page table structure.

Control Register 3 (CR3) anchors the paging structure by giving the high order 20 bits, the page frame address, of the page directory. CR3 is changed upon a task switch through a 386 TSS from an entry in the task state segment. It can also be loaded explicitly.

PROGRAMMING NOTE:

> No limit is given for the first level page directory, in order to simplify the paging mechanism. The maximum size of the first level page directory is one page. To support a first level table smaller than 1 page, the limit fields in the descriptor tables (GDT and LDT) can be set to keep the linear address below a value that permits the directory to occupy less than a page. This is the largest value for the sum (Segment Base address + Segment Limit) in the associated table.

## 3.4.2. Page Table Structure

The mapping of "linear" to physical addresses is described by a two-level page table. The first level is formed by a single page containing the page Directory, which contains 1024 entries, one for each Page table in the mapping. Each directory entry points to a page containing a Page table which contains 1024 entries, one for each page frame in a portion of the mapping. The maximum number of pages per table is 1024x1024=1024K pages, which can map 4 gigabytes.

A subset of the protection attributes are brought out at both the Directory level and the Page level, to permit sharing of second level tables, and to permit sub-structure to be defined within huge segments.

```
                         LINEAR ADDRESS

                    +------+------+--------+
                    | dir  | page | offset |      +-------------+
                    +------+------+--------+      |             |
                       |       |       |          |             |
      -----------------|       |       |          |             |
      |                        |       |          |             |
      |                        |       +--------->|Physical     |
      |                        |                  | Address     |
      |                        |                  |             |
      |                        |                  |             |
      |    +-------------+     |                  +-------------+
      |    |             |     |                  A
      |    |             |     |  +------------+  |
      |    |             |     |  |            |  |
      |    |             |     |  |            |  |
      |    +-------------+     |  +------------+  |
      +--->| Dir. Entry  |---  +->|Pg Tbl Entry|--|
           +-------------+  |     +------------+
           |             |  |     |            |
           +-------------+  |     |            |
           A                |     |            |
+--------+ |                |---->+------------+
| CR3    |--|
+--------+ Page Directory           Page Table      Page Frame
```

Figure 3.4.1 Two Level Page Table Structure

Figure 3.4.1 illustrates the use of the two level table to map linear address to physical addresses by use of the two level mapping table. CR3 roots the table structure by pointing to the first level of the table, the Directory. Bits 31..22 are used as an index into this table to select a 4-byte Page Directory Entry. This Directory entry contains the page address (upper 20 bits) of the second level of the table, the Page Table. Bits 21..12 are used an an index into this table to select a 4-byte Page Table Entry. This Page Table entry contains the page address of the Page Frame which contains the data. Bits 11..0 give the offset of the first byte of the addressed data item within this page frame.

### 3.4.3. Directory and Page Table Entries

This section gives the format of the entries in both levels of the page table structure. The entries at the two levels have the same structure. Bit 0 is the Present Bit. If 1, the entry is valid for address translation. If P=0, the entry is invalid, and the remaining 31 bits are available for software use. In these figures, AVAIL(ABLE) fields are uninterpreted by the hardware and are available for use by software. Bits marked as 0 must be zero (for compatibility with future processors).

Directory Entry

```
 31                                      12 11  9 8 7 6 5 4 3 2 1 0
+---------------------------------------+-----+---+-+-+---+-+-+-+
|                                       |     |   | | |   |U|R| |
|    PAGE TABLE ADDRESS                 |AVAIL|0 0|D|A|0 0|-|-|P|
|                                       |     |   | | |   |S|W| |
+---------------------------------------+-----+---+-+-+---+-+-+-+
```

Directory Entry (Points to Page Table)

Page Table Entry

```
 31                                      12 11    8 7 6 5 4 3 2 1 0
+---------------------------------------+-----+---+-+-+---+-+-+-+
|                                       |     |   | | |   |U|R| |
|    PAGE FRAME ADDRESS 31..12          |AVAIL|0 0|D|A|0 0|-|-|P|
|                                       |     |   | | |   |S|W| |
+---------------------------------------+-----+---+-+-+---+-+-+-+
```

Page Table Entry (Points to Page Frame)

Invalid Directory or Page Table Entry

```
 31                                                            1 0
+-------------------------------------------------------------+-+
|                                                             | |
|                    A V A I L A B L E                        |0|
|                                                             | |
+-------------------------------------------------------------+-+
```

### 3.4.3.1. Present Bit

The P (Present) bit indicates if a Page Directory or Page Table Entry can be used in address translation. P=1 indicates that the Entry can be used in address translation. P=0 indicates that the entry is not valid for address translation, and the rest of the entry is user defined. If P=0, none of the other bits in the entry are tested by the hardware. In particular, if P=0, the U/S and R/W bits are not checked. If P=0 in either the Page Directory or the Page Table Entry used in mapping an address, a not-present fault is raised. A page-level access violation is raised only if P=1 in both levels of the table, and the U/S and R/W bits indicate a violation, as described in the next section.

IMPLEMENTATION NOTE:

> The P bits in both levels of the table must be checked "before" the U/S and R/W bits in either level of the table in order to permit these two bits to be ignored by the hardware in not-present pages. If P=0, the U/S and R/W bits are "don't care" bits to the hardware.

### 3.4.3.2. Page level Protection (U/S, R/W Bits)

The U/S and R/W bits are used to provide User/Supervisor and Read/Write protection for individual pages or all pages covered by a directory entry. The U/S and R/W bits in the second level page table entry apply to the page described by that entry. The U/S and R/W bits in the first level Page Directory Table apply to all pages described by the page table pointed to by that directory entry. The U/S and R/W bits for a given page are obtained by taking the numerically smaller (most restrictive) of the U/S and R/W bits (concatenated as a two-bit number) from the directory table and page table entries used to address that page. The

U/S and R/W bits restrain access to memory as follows:

| U/S | R/W | Permitted Level 3 Access | Permitted Level 0,1,2 Access |
|---|---|---|---|
| 0 | 0 | NONE | Read/Write |
| 0 | 1 | NONE | Read/Write |
| 1 | 0 | Read-only | Read/Write |
| 1 | 1 | Read/Write | Read/Write |

The U/S, R/W setting of 0,0 does not indicate read-only access to supervisor level. The subset of the protection attributes available at the page level does not support read-only supervisor areas. Segmentation protection can be used to prevent supervisor write access to an entire segment, if desired.

The U/S and R/W bits can be used to restrict access to pages to code executing at "supervisor" levels (0, 1, and 2), or to prevent "user" level (3) from writing into a page. If a page has a U/S and R/W bits value of 10, write accesses cause faults if the processor is executing at level 3, but do not cause faults if the processor is at level 0, 1, or 2.

PROGRAMMING NOTE:

> The assignment of "supervisor" levels as 0, 1, and 2, and user level as level 3, permits the page level protection to prevent a program executing in Virtual 8086 mode from accessing code or data areas belonging to levels 0, 1, or 2. Since there is no segment level protection provided in Virtual 8086 Mode, the subset of the protection model provided at the page level (the R/W and U/S bits) were "optimized" towards the needs of Virtual 8086 mode. Since a program in Virtual 8086 mode can generate any linear address (by using the 32-bit extensions), the page level protection must be used to protect supervisor areas from Virtual 8086 mode programs.
>
> Programs executing at levels 0, 1, and 2 will bypass the page level protection, as shown in the above table. The segmentation level can be used to prevent level 2 programs from accessing level 1 or 0 data, (level 1 programs from accessing level 0 memory), as necessary.
>
> The page level protection is useful for providing a "piece" of a level 3 segment that is accessible only at lower levels, to allow an OS to store protected data in the same segment provided to the user level program.

In order to access physical memory through the combination of segmentation and paging, 3 levels of table are traversed. A descriptor table is referenced by the segmentation system, and is used to relocate the virtual address to a linear address. In order to map the linear address to a physical address, two levels of page table are referenced. Each of the three entries in the address translation path contain access rights: the Segment Descriptor Table, the Page Directory, and the Page Table. Both sets of access rights, those in the segment table and those in the page tables, must pass for a reference to be legal. The access rights at the page level permit write-protection to be specified for individual pages, or for all pages covered by a second level Page Table.

The U/S and R/W bits for a page are obtained by combining the U/S and R/W bits from the two levels of the page table. This permits either level to restrict the other. This assists sharing of second level tables, since the second level table can indicate full Read/Write (or User) status for the pages, and the first level entry which points to this second level table can indicate Read-Only (or supervisor) access.

The combination of the page directory and page table entries is described in table 3.1 below. The combined column has an "x" in the R/W bit if the combined U/S bit is 0 (level 0, 1, 2), to inicate that the page level Read/Write attribute is ignored for level 0, 1, or 2 accesses.

| DIRECTORY | | PAGE | | COMBINED | |
|---|---|---|---|---|---|
| U/S | R/W | U/S | R/W | U/S | R/W |
| 0 | 0 | 0 | 0 | 0 | x |
| 0 | 0 | 0 | 1 | 0 | x |
| 0 | 0 | 1 | 0 | 0 | x |
| 0 | 0 | 1 | 1 | 0 | x |
| 0 | 1 | 0 | 0 | 0 | x |
| 0 | 1 | 0 | 1 | 0 | x |
| 0 | 1 | 1 | 0 | 0 | x |
| 0 | 1 | 1 | 1 | 0 | x |
| 1 | 0 | 0 | 0 | 0 | x |
| 1 | 0 | 0 | 1 | 0 | x |
| 1 | 0 | 1 | 0 | 1 | 0 |
| 1 | 0 | 1 | 1 | 1 | 0 |
| 1 | 1 | 0 | 0 | 0 | x |
| 1 | 1 | 0 | 1 | 0 | x |
| 1 | 1 | 1 | 0 | 1 | 0 |
| 1 | 1 | 1 | 1 | 1 | 1 |

PROGRAMMING NOTE:

> As an example of the use of access rights at the page level, it is possible to define a large data segment which has some sub-units which are read-only, and other sub-units which are read-write. In this case, the page directory (or page table) entries for the read-only sub-units would have the U/S and R/W bits x0, indicating no write rights for all the

pages described by that directory entry (or for individual pages). This technique would be used in a UNIX system to define a large data segment, part of which is read only (for shared data, and/or ROMmed constants). This enables UNIX-like systems to define a "flat" data space as one large segment, use flat pointers to address within this "flat" space, but be able to protect shared data, shared files mapped into the virtual space, or supervisor areas.

Access rights at both levels of the page table permit restricted sharing of second level tables to permit a UNIX-like system to share a large section of data (a file or a large shared table) between a number of tasks. By setting the access rights in the first level table, the entire shared unit can be write protected, or read/write access granted. The first level tables are local to a task, and so can specify the access rights on a per-task basis. They point to second level tables, which can be shared.

PROGRAMMING NOTE:

Another way to use the paging structure is to simulate a "page-table-per-segment" addressing mechanism. Since each second level table describes a 4 megabyte memory space, segments up to 4 megabytes could be associated with individual second level tables by aligning the segment base on a 4 megabyte boundary in the virtual space. Then a single first level page table entry, and the associated second level page table, correspond to the segment descriptor entry. With this setup, a single second level page table corresponds exactly to one segment, which can expand in size to 4 megabytes.

### 3.4.3.3. Page Usage Statistics (D, A Bits)

The D (Dirty) and A (Accessed) bits are maintained by the processor hardware to provide statistics about page usage in both levels of the page table. With the exception of the D bit in the Directory, these bits are maintained by the hardware. Directory entries have the D bit undefined.

The Accessed bits in both levels of the page table are guaranteed to be set to 1 before a read or write access to an address covered by the entries occurs. The Dirty bit in the memory resident Page Table Entry (second level) is guaranteed to be set to 1 before a write to an address covered by that page table entry occurs.

PROGRAMMING NOTE:

> Note that the only guarantee on the setting of the Accessed bits is that they are set before a read or write access occurs. They may be set even if no access actually occurs, but such cases will be kept to a minimum. For example, the Accessed bit of a Directory entry can be set even if none of the Accessed bits in the page table described by that Directory entry are set. This can happen if all Accessed bits are zero, and a reference to an address in a not-present page occurs, the Accessed bit in the Directory entry may (or may not...) be set before the Page table entry is examined to determine whether or not it is present.

Page Table caching is not transparent. Operating system software must explicitly flush the page table cache whenever the mapping information is changed by software.

PROGRAMMING NOTE:

> The page table cache can be flushed by (re)loading CR3 after the page table information is modified, or by performing a task switch through a TSS which has a different CR3 image than used by the software which modifies the page table.

Processor generated cycles to update the D and A bits are done with indivisible operations (Read-Modify-Write) which bypass caching to guarantee proper operation in a multi-processor system. Software should use LOCKed operations to modify page table mapping information in multi-processor systems.

IMPLEMENTATION NOTE:

> The implementation must ensure that the memory cycles driven by the processor to set the A and D bits will operate properly in a multiprocessor system. This is done by using a locked RMW operation to set the A and D bits, and bypassing caches when referencing the page table entry.
>
> The implementation is free to "cache" the D bit, so that at most one D bit update is done for a given cached entry if

multiple writes occur to the memory space mapped by that entry.

The "Read" part of the RMW operation to set the D bit will bypass the paging cache to read the Directory and Page Table Entries directly from memory. If either entry has the P bit 0, a not-present fault is signalled. Otherwise, the D bit is set and the "Write" part of an RMW operation writes the updated entry back to memory. If already cached the cached entry may or may not be updated by the entry read from the memory resident Page Table. However, a page fault will occur if the Directory or Page Table Entry is marked not present in the memory image of the page table read to set the D bit.

## 4. INTERRUPTS AND EXCEPTIONS

Interrupts and exceptions alter the normal programmatic flow of execution in order to handle external events, and to report error or exceptional conditions which occur during program execution. An Interrupt is an external event that is reported to the CPU from the external environment in an implementation defined way. An Exception is a condition detected as part of program execution. A software interrupt (e.g. INT n instruction) is treated as an exception. Exceptions are classified as faults, traps, or aborts depending on the way they are reported, and whether or not restart of the instruction causing the exception is supported.

Faults are exceptions that are reported "before" the instruction causing the fault. Faults are either detected before the instruction begins to execute, or during the execution of an instruction. If detected during the execution of an instruction, the fault is reported with the machine state restored to a state that permits the instruction to be restarted. That is, any architecturally visible machine state used as an input operand to the faulting instruction is backed up to the state existing immediately before the faulting instruction began execution. In particular, the CS and EIP values saved when a fault is reported will point to the instruction causing the fault. At a fault, the RF bit is set to one (see ch. 2) in the FLAGS image saved when the fault is reported. (see chapter 7)

A Trap is an exception that is reported at the instruction boundary immediately after the instruction in which the exception was detected. The CS and EIP values stored when a trap is reported point to the instruction dynamically after the instruction causing the trap. If a trap is detected in an instruction that alters the program flow, the value of CS and EIP pushed when the trap handler is entered will reflect the alteration of program flow. For example, if a trap is detected in a JUMP instruction, the CS and EIP values will point to the target of the JUMP, not the instruction after the jump.

An Abort is an exception for which the information pushed at the exception, and the machine state when the exception is reported, does not permit precise location of the instruction causing the exception, or restart of the program causing the exception. Aborts are used to report severe errors, such as hardware errors, or inconsistent or illegal values in system tables.

### 4.1. INTERRUPTS

Interrupts are asynchronous events which are not generated in response to the execution of any particular instruction. They are, however, constrained to occur in a fashion that permits the interrupted program to be restarted. Interrupts are accepted at the boundary between instructions. That is, an interrupt is serviced after one instruction is complete, and before the next instruction begins execution. After the interrupt has been serviced, an IRET causes the program to resume at the next instruction. REPeated string instructions are defined to have interrupt windows so that an interrupt can be accepted after each repeat step. This reduces the interrupt latency.

There are 2 kinds of interrupt sources defined by the architecture: NMI, and vectored INTerrupts. An NMI (Non-Maskable Interrupt) is signalled in an implementation dependent way, and causes the processor to vector through interrupt vector 2. The vectored INTerrupts are signalled in an implementation dependent way that indicates the vector to be used to process the interrupt.

The processor is defined to be executing in NMI mode after the receipt of an NMI interrupt, and until an IRET instruction is executed. While executing in NMI mode, NMI interrupts are disabled.

The IF bit in the EFLAGS register can be used to inhibit the reception of external INTerrupts. If IF=0, INTR interrupts are ignored (held pending). If IF=1, INTR interrupts are accepted and will interrupt the currently executing program.

Table 4.1 lists the set of interrupts defined by the 386 architecture. For each interrupt, the interrupt vector location assigned to service the trap is listed. Interrupts use the interrupt servicing sequence described in a later section, and do not push any error codes or other information on the stack.

| Interrupt Number | Function |
|---|---|
| 0 | Intel Reserved |
| 2 | Non-Maskable Interrupt (NMI) |
| 3 | Intel Reserved |
| . | . |
| . | . |
| . | . |
| 31 | Intel Reserved |
| 32-255 | INTR |

Table 4.1 386 Interrupt Vector Assignment

The NMI interrupt has a preassigned vector number of 2. The INTR interrupt reads a vector number (0..255) from external hardware. The vectors listed in the table are reserved by Intel. Many of the reserved vectors are assigned to exceptions, as detailed in the next section.

## 4.2. EXCEPTIONS

Table 4.2 summarizes the exceptions recognized by the 386:

| Function | Interrupt Number | Instruction Which May Generate The Interrupt | Exception Type |
|---|---|---|---|
| Divide error | 0 | DIV, IDIV | FAULT |
| Debug Exceptions | 1 | any instruction | FAULT/TRAP*4 |
| One byte INT | 3 | INT | TRAP |
| Interrupt on overflow | 4 | INTO | TRAP |
| Array bounds check | 5 | CHECK | FAULT |
| Invalid Op-code | 6 | Any illegal instruction | FAULT |
| Device not available | 7 | ESC, WAIT | FAULT |
| System Error | 8 | Any instruction that can generate an exception | ABORT |
| Invalid TSS | 10 | JMP, CALL, any interrupt, IRET | FAULT*1 |
| Segment/Gate not present | 11 | Any segment register modifier | FAULT |
| Stack fault | 12 | Any instruction which references memory through SS | FAULT |
| General purpose protection fault instruction, or code fetch | 13 | Any memory reference | FAULT/ABORT*2 |
| Page Fault | 14 | Any memory reference instruction, or code fetch | FAULT |
| Coproc. error | 16 | ESC, WAIT | FAULT*3 |
| Two byte interrupt | 0-255 | INT N | TRAP |

Table 4.2 Exceptions

NOTES:

(1) Invalid TSS faults are not restartable if they occur during the processing of an external interrupt.

(2) All GP protection faults are restartable. If the fault occurs while attempting to vector to the handler for an external interrupt, the interrupted program is restartable, but the interrupt may be lost.

(3) Coprocessor Errors will be reported as a fault on the first ESC instruction executed after the ESC causing the error.

(4) A debug exception may report both traps on the previous instruction, and faults on the next instruction. Refer to chapter 7 for details.

## 4.3. PRIORITIZING INTERRUPTS AND EXCEPTIONS

Two kinds of priorities are associated with interrupts. Certain processor modes, and certain flag settings will inhibit the signalling of certain interrupts and exceptions. This priority resolution determines whether an interrupt signal, or an exception condition, will be signalled at a given instruction boundary. This is covered in section 4.3.1. The second kind of priority resolves which interrupt to signal if more than one are present and of sufficient priority to be signalled.

### 4.3.1. Enabling/Disabling

As mentioned above, the processor ignores NMI interrupts while the processor is in NMI mode. NMI interrupts are masked at entry the the NMI handler, and remain masked until the next IRET is executed.

In all modes, the IF bit in EFLAGS controls the acceptance of vectored interrupts. If IF=0, vectored interrupts are inhibited. If IF=1, vectored interrupts are enabled.

In all modes, the RF bit in EFLAGS controls the acceptance of debug faults (but not debug traps). This permits debug faults to be raised for a given instruction at most once, no matter how many times the instruction is restarted.

MOV SS, POP SS

In order to suppress interrupts after a Move to Stack Segment or a Pop to Stack Segment instruction in a way that is compatible with the 8086 and 286, these instructions are defined to ignore external interrupts (NMI and INTR), debug exceptions, and single-step interrupts at the next instruction boundary. This will inhibit all of the interrupt sources that were inhibited on the 8086 and 286 on the instruction boundary following the loading of the SS register. Exceptions defined on the 286 can still occur at this boundary (e.g. a segment limit fault on the following instruction), and some new exceptions (e.g. page fault) can also occur at this boundary.

PROGRAMMING NOTE:

It is now possible to get a page fault between the execution of the instruction which loads SS and the next instruction.

Page faults can occur on the instruction following an instruction which loads SS. However, if the processor is in protected mode the stack is undefined only if the page fault handler is invoked by a TRAP or INTERRUPT gate, and the page fault occurs at the same privilege level as the page fault handler. In all other cases the handling of the page fault will cause a stack switch to a well defined stack. Systems which implement paging, and which handle page faults within the faulting task (with TRAP or INTERRUPT gates), should take care to ensure that software that executes at the same privilege level as the page fault handler uses the new LSS instruction rather than the MOV SS, xxx; MOV SP, xxx instruction pair to initialize a new stack. In all cases, the trap handler for page faults will execute at level 0, limiting the scope of the problem to OS kernal code. Ensuring the integrity of systems where the page fault handler executes at a level other than zero, or is a conforming procedure, is an exercise left to the reader.

In particular, if the processor is executing in Virtual 8086 mode a page fault occuring after a move to SS will always do a stack switch as part of fielding the page fault.

### 4.3.2. Simultaneous Priority

The second kind of interrupt priority is used to select the interrupt or exception to signal at an instruction boundary if more than one is present and of sufficient priority to be signalled. This type of priority is used to decide which interrupt or exception to signal if several are detected simultaneously. From an architectural viewpoint, "simultaneously" means on the same instruction, and has nothing to do with the clock in which the exception is reported. In other words, the architectural definition assumes all instructions take just one indivisible "clock" period.

| Priority | Interrupt/Exception |
|---|---|
| 1 | Exception Faults |
| 2 | TRAP instructions |
| 3 | Debug Traps for this instr. |
| 4 | Debug Faults for next instr. |
| 5 | NMI interrupt |
| 6 | INTR interrupt |

Table 4.3 Simultaneous Interrupt/Exception Priorities

Segment faults are detected before page faults on a given memory reference. That is, if a memory reference will produce both a segment fault and a page fault, the segment fault will be reported. Instruction fetches are treated as a series of single byte references, and the exception for the first byte with a problem is reported. If an instruction fetch byte will produce both a segment fault and a page fault, the segment fault is reported.

Debug faults on the "next" instruction are prioritized higher than external interrupts in order to permit them to be processed at the same priority level as debug traps on the "current" instruction. Providing different priority levels for debug faults vs. traps is an unnecessary complication.

## 4.4. INTERRUPT/EXCEPTION SIGNALLING

Signalling an interrupt or exception involves saving the state of the current computation, and transferring control to an interrupt handler. For some exceptions, information is passed on the stack to the handler. After completing the handling of an interrupt or exception, the handler resumes the interrupted computation by popping any error codes and executing an IRET instruction.

The sequence of actions taken by the processor to signal an interrupt or exception (initiate a handler) depends on the mode in which the processor is executing. If the processor is in REAL mode, interrupts and exceptions are signalled as on the 8086, by vectoring through an interrupt procedure list located at the base address in the IDTR register (initialized to 0). If the processor is in protected mode, interrupts and exceptions are signalled as on the 286, by vectoring through an Interrupt Descriptor Table (IDT). Virtual 8086 mode is treated as a subset of protected mode.

### 4.4.1. REAL mode Signalling

An interrupt or exception signalled in REAL mode uses exactly the 8086 sequence, in pseudo-assembly language:

```
PUSH      FLAGS  ; Push low 16 bits of EFLAGS
IF := 0;
TF := 0;
PUSH      CS     ; 16-bit push
PUSH      IP     ; 16-bit push
; No error codes are pushed.
JMPf      DWORD PTR IDTR:[int*4] ; FAR jump through table.
```

No error codes are pushed in REAL mode for any of the pseudo-protection faults generated in that mode, even though the corresponding fault in protected mode will push an error code.

Interrupts are always handled using the same sequence as on the 16-bit processors, saving only the low 16 bits of EFLAGS and EIP. This means that REAL mode code cannot use any of the 386 extensions to CODE segment size, since executing more than 64K past the CS base would require use of the upper bits of EIP, which are not saved at an interrupt in REAL mode.

### 4.4.2. Protected Mode Signalling

In protected mode, interrupt signalling is controlled by the Interrupt Descriptor Table (IDT). To signal an interrupt, the descriptor indexed by the interrupt or exception number is retrieved from the table and used to locate the interrupt handler. This descriptor must be of one of the following descriptor types:

- TASK GATE (type 5)
- 286 INTERRUPT GATE (type 6)
- 286 TRAP GATE (type 7)
- 386 INTERRUPT GATE (type 14)
- 386 TRAP GATE (type 15)

A TASK GATE indicates that the interrupt is signalled by dispatching the task described by the TSS pointed to by the gate. The normal task switching mechanism is used. The TSS can be either a 286 or a 386 TSS, with the task switch reflecting the differences in task switching described elsewhere.

An interrupt signalled through a 286 INTERRUPT or TRAP GATE will operate exactly as on the 286. Intra-level traps will push the 16-bit FLAGS, CS, and IP (lower 16-bits of EIP), and then jump to the beginning of the interrupt handler. Inter-level traps will switch to a new stack, and then push the old SS, the old SP (16 bits), the old FLAGS (16 bits), old CS, and old IP (16 bits), and then jump to the beginning of the interrupt handler. If an error code (16 bits) is supplied, it is pushed after IP is pushed, but before the jump to the interrupt handler. An interrupt signalled through a 386 INTERRUPT or TRAP GATE will push the full 32-bit ESP, EFLAGS, and EIP registers, and will push the SS and CS selectors and error codes padded to 32-bits with high-order bits undefined, in order to keep the stack aligned at a 4-byte boundary. Otherwise, the same sequence of actions occurs as for the 286 gates.

Once control has transferred to the handler software (which may include a stack and/or task switch), error codes are pushed on the stack. Error codes are defined for certain exceptions, and are pushed to provide information about the exception to the handler for that exception. Interrupts and exceptions handled by a 286 Trap or Interrupt Gate, or by a Task Gate through a 286 TSS, will push a 16-bit error code. Interrupts and exceptions handled by a 386 Trap or Interrupt Gate, or by a Task Gate through a 386 TSS, will push the error code as a 32-bit value containing the code in the lower 16 bits, with the high-order 16 bits undefined. Error codes must be popped off the stack before executing the IRET instruction to return from the interrupt handler.

The IRET instruction is used to return from an interrupt. There are two forms of IRET: one with a 16-bit operand size (the 286 compatible form), and one with a 32-bit operand size. For either the 16 or 32-bit IRET, if the NT flag is set, a task switch to the task linked to the current task's TSS is performed. Otherwise an intra or inter-level return is performed by popping the register images from the stack that were pushed when the interrupt routine was initiated.

The 16-bit IRET will unwind the actions of a 286 INTERRUPT or TRAP GATE by popping IP (16-bits), CS, and FLAGS (16 bits). Then, if the RPL of the CS selector popped from the stack is different from CPL, an inter-level IRET is performed by popping SP (16-bits) and SS, restoring the stack for the interrupted level, and then changing CPL to the RPL of the CS selector. Error codes pushed on the stack when the interrupt was serviced must be popped off the stack before the IRET instruction is executed.

The 32-bit IRET will unwind the actions of a 386 INTERRUPT or TRAP GATE by popping EIP (32-bits), CS, and EFLAGS (32 bits). Then, if the RPL of the CS selector popped from the stack is different from CPL, an inter-level IRET is performed by popping ESP (32-bits) and SS, restoring the stack for the interrupted level, and then changing CPL to the RPL of the CS selector. The SS and CS selectors are popped as 32-bit words, with the selector value taken from the lower 16 bits only. Error codes pushed on the stack when the interrupt was serviced must be popped off the stack before the IRET instruction is executed.

### 4.4.3. Virtual 8086 Signalling

Interrupts in Virtual 8086 mode are signalled as in Protected mode, by vectoring through the IDT. One additional effect is that the interrupt handler is executed in Protected mode, not in Virtual 8086 mode, so the trap will change stacks even if there is no level change, in order to change modes.

Chapter 5 contains more information on interrupts in REAL and Virtual 8086 modes.

## 4.5. PAGE FAULT HANDLING

Page faults will be signalled through interrupt vector 14, which was unassigned on the 286. After the fault is accepted, and control transferred to the fault handler, information about the fault is available in the error code pushed at entry to the interrupt routine, and in control register 2. CR2 contains the linear address used in the access that raised the fault. See chapter 2 for more details.

Page faults are constrained to occur only after all instructions preceding the instruction causing the page fault have completed successfully.

IMPLEMENTATION NOTE:

> Implementations which overlap instruction fetch or operand fetch with instruction execution must not report page faults which occur while prefetching until all preceding instructions have completed successfully.

Page faults which occur at certain times when accessing certain system objects are not recoverable. These will be signalled with a system error abort (vector 8) rather than a page fault. The operating system can ensure that no page faults will occur in the middle of accessing the affected system objects.

### 4.5.1. Page Fault Signalling

When a page violation is detected, the page fault handler is signalled by vectoring through interrupt vector 14. The program currently executing is faulted in a fashion that permits restarting after the page fault is serviced. The CS and EIP register images saved at the fault will point to the instruction causing the page fault. CR2 is loaded with the linear address causing the page fault.

#### 4.5.1.1. Error Code Format

The error code pushed as part of handling the page fault contains status bits indicating the cause of the page fault. The error code has the following format, which is similar to the low order bits of the page table entry. The field names are similar, but they serve different functions. The page table entries specify the operations that are to be restricted, wheras the error code specifies the type of access that was attempted.

The error code is 16 bits. 286 Trap and Interrupt gates, and Task gates through a 286 TSS will push a 16-bit error code. 386 Trap and Interrupt gates, and Task gates through a 386 TSS will push a 32-bit error code, with this 16-bit error code value stored in the low order 16 bits, and the upper 32 bits undefined.

```
 15                         3 2 1 0
+--------------------------------+
|                          |U|W| |
|         UNDEFINED        |-|-|P|
|                          |S|R| |
+--------------------------------+
```

Page Fault Error Code Format

U/S# The U/S# bit indicates whether the access causing the fault originated when the processor was executing in User Mode (U/S#=1), or Supervisor mode (U/S#=0).

W/R# The W/R# bit indicates whether the access causing the fault was a Read (W/R#=0), or a Write (W/R#=1).

The U/S# bit and W/R# bits indicate the type of access which caused the page fault, as given in the following table:

| U/S# | W/R# | ACCESS TYPE |
|---|---|---|
| 0 | 0 | Supervisor* Read |
| 0 | 1 | Supervisor Write |
| 1 | 0 | User Read |
| 1 | 1 | User Write |

* Descriptor table accesses will fault with U/S#=0, even if the program is executing at level 3.

P The P bit indicates whether a page fault was caused by a not-present page (P=0), or by a page level protection violation (P=1).

## 4.6. SYSTEM ERRORS

It is possible for a second page fault or segment fault to be generated while the processor is attempting to signal another. For example, the TSS for a page fault handler signalled through a Task Gate may be located at a relocated address that has a page marked not present. Or, the not present (segment) handler may have its descriptor marked not present. Rather than attempt to continue in these cases, the processor attempts to invoke the system error handler by vectoring through interrupt vector 8.

An interrupt 8 is generated for system error aborts, with an error code of 0 pushed. Instruction restart is not supported. This interrupt is not generated in REAL mode.

For double fault detection, exceptions are grouped into 3 categories:

(1) benign exceptions

(2) trapping exceptions

(3) page faults

The 286 lumped all exceptions into two categories when considering double faults. Zero-divide errors (interrupt 0), and segment exceptions (interrupts 10, 11, 12, and 13) are considered as trapping exceptions, and count towards double faults. All other interrupts and exceptions (types 1, 2, 3, 4, 5, 6, 7, 9, and 16) did not count towards double faults on the 286, and do not on the 386. The type 9 interrupt can no longer occur on the 386.

For example, a segment fault occuring while attempting to invoke a type 4 interrupt as part of the INTO instruction would not cause a double fault, but would instead report the segment fault. But, a segment fault occuring while attempting to invoke the zero-divide handler would raise a double fault. The 386 carries this category forward, and adds page faults as a separate category to support paging under segmentation.

Two double fault types are used to detect recursive errors which occur while attempting to signal an exception. If a segment fault occurs while trying to signal a segment fault, a system error is signalled. If a page fault or segment fault occurs while trying to signal a page fault, a system error is raised.

If a page fault occurs while attempting to signal a segment fault, no system error is signalled, and the page fault takes precedence over the segment fault. This supports the hierarchy of segmentation and paging, with paging "underneath" segmentation, by allowing the segmentation support to be independent of the paging support. In particular, the segmentation fault handlers need not be concerned about paging or page faults, but can be implemented on top of paging.

If a page or segment fault is generated while attempting to invoke the system error abort handler, the processor halts.

Double fault checking covers all data references generated to handle the fault, such as pushing information onto a stack, and TSS references, along with segment checks on the validity of the CODE segment containing the interrupt handler, and the offset of the entry point of the interrupt handler. Page faults at the entry point of the handler will not cause double faults, since these are signalled after the processor context has changed to that of the interrupt handler.

## 4.7. Restartability Implications on Software

All instructions are restartable after a page or segment fault on the 386. Operating system software need not participate in the restart backout process, as the machine will report a page or segment fault with the machine in a state that permits clean restart of the faulting instruction after the condition which raised the fault has been corrected. The CS:EIP pushed at a fault will point to the instruction which faulted, and which needs to be restarted. After the OS has corrected the cause of the fault, the faulting instruction can be restarted simply by executing an IRET instruction to terminate the fault handler.

It is not necessary for restartability purposes for the machine to raise a fault such that the machine state is exactly that existing just before the faulting instruction was executed. It is only necessary to fault with input machine state, and machine state unaffected by the faulting instruction, exactly that existing just before executing the faulting instruction. Machine state that is an output of the faulting instruction may be partially modified when a fault is raised, provided that such output machine state is not also used as an input to the instruction.

Instruction restart is not guaranteed if an input operand is located (or aliased) below any of the current stack pointers. This includes the current SS:ESP address, and the inner level stack pointers located in the current TSS. The memory areas below (at lower addresses) than the current stack pointer(s) may be modified as part of interrupt and exception processing, so use of these memory areas as input operands to instructions will produce results that are predictable only so far as the interrupt activity is predictable.

Instruction restart is not guaranteed if an instruction causes a task switch to a TSS that is located in a not-present page. Operating system software must guarantee that the TSSs for the tasks that can be hardware dispatched are locked into pages in physical memory in order to avoid restart problems due to page faults when accessing a TSS.

Other than the above two constraints, volitile stack areas and Task Switch operations, instruction restart is guaranteed for all instructions, and is accomplished without intervention from software.

### 4.7.1. Partial Modification of Machine State

Machine state which does not affect the correct execution of the faulting instruction need not be backed up to the state existing immediately before the faulting instruction began execution. The only constraint on modifying machine state is that no instruction will modify any machine state that is one of its input operands unless no faults can occur on that instruction. The processor will refrain from modifying any input machine state until it has verified that the instruction can complete without raising faults.

For example, the CF (Carry Flag) is an input operand to the ADC (ADd with Carry) and SBB (SuBtract with Borrow) instructions. When implementing these instructions, the machine must check that no faults can occur before the CF flag is modified. For the SBB and ADC instructions with memory destinations, this requires that the processor delay in updating CF until after the write to the destination has been verified to complete fault-free.

However, machine state which is not an input operand to an instruction can be modified by the processor, even if this results in the state being modified when a page or segment fault occurs. For example, the ADD instruction with a memory destination may change CF before it verifies that the write to the destination field can occur without errors. Another example is that PUSHA (Push all) can modify memory by "pushing" some registers even if the instruction cannot complete without raising a fault. In this case, memory below the current stack pointer will be modified when the fault handler is entered. However, PUSHA must not update the ESP register unless it has guaranteed that no faults can occur, since ESP is an input operand.

### 4.7.2. Aliasing of Operands

Two addresses are aliased if they are equal in the virtual, linear, or physical address spaces. Aliasing of memory operands will not permit instruction restart unless the instruction is defined for overlapping memory operands. In particular, since the MOVSw instruction is defined for overlapping memory operands, full restart on faults is supported with this instruction, even if its operands overlap either in the virtual, linear, or physical address spaces. However, since the MOVSd instruction is undefined for overlapping memory operands, restart of this instruction is supported only if the operands do not overlap in the virtual space, and do not overlap through aliasing in the linear or physical spaces.

### 4.7.3. Stack Areas

The memory locations immediately below the stack pointer (and those below the inner level stack pointers stored in the current TSS) may be partially modified by instructions which raise page or segment faults, even if these areas are input operands to the instruction. Aliasing or overlap of instruction operands with the volitile memory area below the current stack pointer(s) is not supported. The current stack, as well as the inner level stacks (initial stack pointers stored in the current TSS) must not be used as an input operand, or aliased with other memory areas in the linear or physical spaces that are used as input operands. Restart is not guaranteed for instructions which have input operands which are in or are aliased to these areas.

The ENTER instruction is not restartable if it is executed with EBP<ESP (BP<SP for 16-bit operand size), and level >= 2 (no stack copies are performed unless level >=2). ENTER is intended to be used for stack bookkeeping at procedure entry, when EBP points to the previous stack frame, which must be at a higher address than the current top of stack. If ENTER is executed with EBP<ESP and level >= 2, it violates the constraint against having input operands below the top of stack.

Calling through a CALL gate where the old stack is aliased below the new stack is not restartable. In this case, the copying of parameters from the old to the new stack will not restart properly, since a fault may occur after some of the parameter words have been copied from the old stack, but before all parameter words have been copied.

Calling through a CALL gate where the new stack is aliased on top of the descriptor table entry for the gate or for the code or stack segments is not restartable. In this case, "pushing" CS:EIP or copying parameters to the new stack may overwrite the descriptor table entry for the call gate, so that if a fault occurs before the CALL completes, restarting the CALL will result in use of the partially overwritten descriptor table entries.

# 5. EXECUTING 8086 CODE

## 5.1. Introduction

Two operating modes of the 386 are bit compatible with the 8086/186 instruction set. These are provided for executing 8086 object code. REAL mode is provided just as in the 286, and is the mode used by the processor after a RESET. Virtual 8086 mode is a subset of 386 protected mode, and permits 8086 code to be executed within the protected and paged operating environment provided on the 386. These two modes have much in common, with the only difference outside of the protection model being the way interrupts are handled in the two modes.

Both modes will execute 8086 object code with 286 REAL Mode semantics. There are a number of minor differences between 8086 semantics and 286 REAL Mode semantics, summarized in [7].

Virtual 8086 mode relies on a virtual machine monitor which executes in 386 protected mode to emulate a few privileged and sensitive instructions which cause protection traps in virtual 8086 mode.

Both modes form addresses and treat segment registers in the same way. In both modes, segment relocation is performed as in the 8086: the 16-bit value in the segment registers is shifted left by 4 bits and added to the 16 or 32-bit effective address to obtain a 32-bit linear address. For full compatibility with 286 REAL mode, pseudo-protection faults (int. 12 or 13 with no error code) will occur if an address outside the range 0..65535 is generated. Therefore, although the 32-bit address modes are available to generate 32-bit offsets, they are of limited use since they must be limited to a 64K range to avoid pseudo-protection faults. Also, this implies that REAL and Virtual 8086 modes can address at most the range addressed by the 286 in REAL mode: from 0 to 1 Meg plus 65519 (0..10FFEF hex).

The set of instructions that cause undefined opcode traps (interrupt 6) is the same in both modes. This includes a set of instructions that are recognized in protected mode which manipulate or interrogate 286 selector entries, such as VERR, VERW, ARPL, etc. This set is the same set identified for the 286.

The set of privileged (execute at level 0 only) and sensitive (execute at levels less than or equal to IOPL) instructions are defined for both modes. In addition to the privileged and sensitive instructions defined for Protected mode, several instructions are made sensitive in REAL and Virtual 8086 modes to assist full emulation of 8086 programs. Note that since REAL mode executes at privilege level 0, none of the privileged or sensitive instructions will cause faults in REAL mode.

Both modes are able to execute the instruction set extensions described in chapter 6.

The major difference between the two 8086 modes is the way that interrupts are handled. In REAL mode, interrupts trap through a vector of procedure pointers located at the base address in IDTR. Each entry is 4 bytes long, and gives the starting address of the interrupt handling procedure. In Virtual 8086 mode, interrupts are vectored through the protected mode IDT, and exit virtual 8086 mode to handle the interrupt in "native" 386 mode. These 386 trap handlers can either completely process the interrupt themselves, and simply return to virtual 8086 mode with an IRET, or can "reflect" the interrupt to the virtual 8086 mode program by manipulating the virtual 8086 state saved at the trap.

Some traps are defined to push error codes on the new stack as part of the trap handling process if the processor is in protected mode. Interrupts fielded while the processor is executing in virtual 8086 mode will be handled in protected mode (a mode switch occurs as part of interrupt handling). Consequently, traps fielded while the processor is executing in virtual 8086 mode will push error codes. Traps fielded in REAL mode will not push error codes.

REAL mode executes with the PE bit (in the MSW register) set to 0. Virtual 8086 mode executes with the PE bit set to 1. Memory mapping in virtual 8086 mode is supplied by paging.

REAL mode always executes at privilege level 0, Virtual 8086 mode always executes at privilege level 3.

## 5.2. 8086 MODE ADDRESSING

Both modes use the 8086 interpretation of the segment registers rather than the 286/386 protected mode interpretation. To form an address, the value in the programmer-visible segment register is shifted left 4 and added to the 16 or 32-bit effective address to obtain a 32-bit linear address.

COMPATIBILITY NOTE:

> The 386 does not wrap addresses at 1 Megabyte in either REAL mode or Virtual 8086 mode. However, it is possible to simulate the 1 Meg wraparound in Virtual 8086 mode by using paging to alias the 64K bytes above 1 Meg to the 64K bytes starting at linear address 0. This permits the 386 to be compatible with the 8086, with 1 Meg wraparound, and with the 286, with no wrap at 1 meg.

The pseudo-protection checks performed by the 286 are also done on the 386 in REAL and Virtual 8086 modes. Word accesses to offset 0FFFF will cause a pseudo-protection trap, as will pushes with SP=1, and POPs with SP=0FFFF. Use of 32-bit address size to generate effective addresses outside the range 0..0FFFF will also cause pseudo-protection faults.

Implicit use of the stack pointer in PUSH, POP, CALL, RETurn, etc., instructions will use the 16-bit SP register to form a 16-bit effective address. This is identical to the way implicit stack references are handled in Protected mode if the B bit is 0 (the 286 compatible setting) in descriptors loaded into the SS register. This assures compatibility with 286 REAL mode, including the signalling of Stack overflow pseudo-segment limit faults if a push operation is attempted with SP=1 (or a pop operation with SP=0FFFF).

The 32-bit EIP register is used for implicit references to code, just as in protected mode. It is possible to use the 32-bit control transfers while in REAL mode. However, since interrupts will store only the low order 16 bits of EIP an interrupt received while the upper bits of EIP are not 0 will have unpredictable results. It is reccomended that software tools provide a method to flag and/or prevent use of 32-bit control transfers in code destined to be run in REAL mode. Code segment overflow pseudo-protection faults are generated if sequential execution crosses a 64K boundary.

COMPATIBILITY NOTE:

> Control transfer instructions are compatible with both the 8086 and 286, but sequential execution beyond the "end" (64K) of a code segment is compatible with the 286 but not the 8086. The 8086 will wrap around to fetch code from the byte at offset 0FFFF in the segment to offset 0 in the segment. The 286 and the 386 will raise a pseudo-protection exception.

IMPLEMENTATION NOTE:

> Addressing in the two 8086 modes can be implemented in the same fashion as REAL mode addressing on the 286. The same segment relocation hardware, utilizing hidden base, limit, and access rights registers, can be shared with Protected 386 mode. The distinction between 8086 modes and protected mode is handled when segment register values are loaded, rather than at each memory reference. As in the 286, the 8086 modes can load the hidden descriptor registers as follows. The base register can be loaded with a value obtained by shifting the segment register value left 4. The limit register can be loaded with 0000FFFF. The access rights for all segments permit read, write, and execute access (or disable access rights checks). The B bit in SS descriptors is set to 0 to force use of 16-bit effective addressing for implicit stack references. And, the D bit in the CS descriptor can be set to 0 to indicate the 16-bit default for code.

Because all of the segment registers are reloaded with 8086 values when the processor enters virtual 8086 mode, there is no possiblity of having "stale" data in segment registers or hidden descriptors.

## 5.3. UNDEFINED OPCODE TRAPS

As in 286 REAL mode, the two 8086 modes will generate undefined opcode traps (interrupt 6) if certain instructions specific to protected mode are executed. These are:

ARPL
LAR
LSL
VERR
VERW
STR
LTR
SLDT
LLDT

## 5.4. PRIVILEGED AND SENSITIVE INSTRUCTIONS

Programs execute at privilege level 3 when in Virtual 8086 mode. A program executing in Virtual 8086 mode is subject to all of the level protection checks defined in protected mode. Attempts to execute privileged instructions will cause GP faults with 0 error code. Sensitive instructions, as defined on the 286, will also cause GP faults with 0 error code if the I/O Privilege Level (given by the IOPL field in the FLAGS) is less than 3.

The INT n, PUSHF, POPF, and IRET instructions are made IOPL sensitive instructions if and only if the processor is executing in Virtual 8086 mode. Note that only INT n is made sensitive, the INTO and INT3 instructions are not sensitive, and will trap directly through their respective interrupt vectors.

The IOPL sensitive instructions can also be considered IOPL sensitive in REAL mode, but since REAL mode executes at privilege level 0, no IOPL sensitive instruction faults can be generated in REAL mode.

PROGRAMMING NOTE:

> A Virtual Machine Monitor supporting a task executing in Virtual 8086 mode can use the general protection violation fault to emulate the INT n, PUSHF, POPF, and IRET instructions. An INT n (PUSHF, POPF, or IRET) instruction executed in Virtual 8086 mode will raise a privileged instruction fault (assuming it is not at IOPL). The fault handler can a) determine if the fault was raised by a program in Virtual 8086 mode by examining the VM bit in the FLAGS image stored when the fault is taken, and b) examine the instruction pointed to by CS:EIP to determine if it is an INT n (PUSHF, POPF, or IRET) instruction. The fault handler can reflect

the INT n back to the 8086 program by manipulating the 8086 register images stored when the protection fault is received, or the interrupt can be translated to protected mode interrupts, CALLs to OS services, etc. The other sensitive instructions can be interpreted in the monitor in a similar fashion.

## 5.5. ENTERING AND LEAVING REAL MODE

REAL mode is entered at RESET. REAL mode is exited by executing a LMSW or MOV CR0 instruction with an operand that has a 1 in the PE bit position (bit 0). While executing in REAL mode, the PE bit in CR0 is zero.

## 5.6. ENTERING AND LEAVING VIRTUAL 8086 MODE

Virtual 8086 mode is entered by executing an IRET instruction (at CPL=0) or Task Switch (at any CPL) with a FLAGS image containing a 1 in the VM bit position while the processor is executing in Protected Mode. That is, one way to enter Virtual 8086 mode is to switch to a task with a 386 TSS that has a 1 in the VM bit in the EFLAGS image. The other way is to execute a 32-bit IRET instruction at privilege level 0, where the stack has a 1 in the VM bit in the EFLAGS image. POPF does not affect the VM bit, even if the processor is in Protected Mode or level 0, and so cannot be used to enter Virtual 8086 Mode. PUSHF always pushes a 0 in the VM bit, even if the processor is in Virtual 8086 Mode, so that a program cannot tell if it is executing in REAL mode, or in Virtual 8086 mode.

The VM bit can be set only by executing an IRET instruction at privilege level 0, or by any instruction or Interrupt which does a task switch in Protected Mode (with VM=1 in the new FLAGS image), and can be cleared only by an interrupt or execption in Virtual 8086 Mode. IRET and POPF instructions executed in REAL mode or Virtual 8086 mode will not change the value in the VM bit.

The transition out of virtual 8086 mode to 386 protected mode occurs only on receipt of an interrupt or exception. In Virtual 8086 mode, all interrupts and exceptions vector through the protected mode IDT, and enter an interrupt handler in protected 386 mode. That is, as part of interrupt processing the VM bit is cleared.

Because the matching IRET must occur from level 0, if an Interrupt or Trap Gate is used to field an interrupt or exception out of Virtual 8086 mode, the Gate must perform an inter-level interrupt only to level 0. Interrupt or Trap Gates through conforming segments, or through segments with DPL>0, will raise a GP fault with the CS selector as the error code.

### 5.6.1. Task Switches to/from Virtual 8086 mode

Tasks which can execute in virtual 8086 mode must be described by a TSS with the new 386 format (TYPE 9 or 11 descriptor).

A task switch out of virtual 8086 mode will operate exactly the same as any other task switch out of a task with a 386 TSS. All of the programmer visible state, including the FLAGS register with the VM bit set to 1, is stored in the TSS. The segment registers in the TSS will contain 8086 segment base values rather than selectors.

A task switch into a task described by a 386 TSS will have an additional check to determine if the incoming task should be resumed in virtual 8086 mode. Tasks described by 286 format TSSs cannot be resumed in virtual 8086 mode, so no check is required there (the FLAGS image in 286 format TSS has only the low order 16 FLAGS bits). Before loading the segment register images from a 386 TSS, the FLAGS image is loaded, so that the segment registers are loaded from the TSS image as 8086 segment base values. The task is now ready to resume in virtual 8086 execution mode.

### 5.6.2. Transitions Through Trap and Interrupt Gates, and IRET

A task switch is one way to enter or exit virtual 8086 mode. The other method is to exit through a Trap or Interrupt gate, as part of handling an interrupt, and to enter as part of executing an IRET instruction. The transition out must use a 386 Trap Gate (Type 14), or Interrupt Gate (Type 15), which must point to a non-conforming level 0 segment (DPL=0) in order to permit the trap handler to IRET back to the Virtual 8086 program. The Gate must point to a non-conforming level 0 segment to perform a level switch to level 0 so that the matching IRET can change the VM bit. 386 gates must be used, since 286 gates save only the low 16 bits of the FLAGS register, so that the VM bit will not be saved on transitions through the 286 gates. Also, the 16-bit IRET (presumably) used to terminate the 286 interrupt handler will pop only the lower 16 bits from FLAGS, and will not affect the VM bit. The action taken for a 386 Trap or Interrupt gate if an interrupt occurs while the task is executing in virtual 8086 mode is given by the following sequence.

(1) Save the FLAGS register in a temp to push later. Turn off the VM and TF bits, and if the interrupt is serviced by an Interrupt Gate, turn off IF also.

(2) Interrupt and Trap gates must perform a level switch from 3 (where the VM86 program executes) to level 0 (so IRET can return). This process involves a stack switch to the stack given in the TSS for privilege level 0. Save the Virtual 8086 Mode SS and ESP registers to push in a later step. The segment register load of SS will be done as a Protected Mode segment load, since the VM bit was turned off above.

(3) Push the 8086 segment register values onto the new stack, in the order: GS, FS, DS, ES. These are pushed as 32-bit quantities, with undefined values in the upper 16 bits. Then load these 4 registers with null selectors (0).

(4) Push the old 8086 stack pointer onto the new stack by pushing the SS register (as 32-bits, high bits undefined), then pushing the 32-bit ESP register saved above.

(5) Push the 32-bit FLAGS register saved in step 1.

(6) Push the old 8086 instruction pointer onto the new stack by pushing the CS register (as 32-bits, high bits undefined), then pushing the 32-bit EIP register.

(7) Load up the new CS:EIP value from the interrupt gate, and begin execution of the interrupt routine in protected 386 mode.

The transition out of virtual 8086 mode performs a level change and stack switch, in addition to changing back to protected mode. In addition, all of the 8086 segment register images are stored on the stack (behind the SS:ESP image), and then loaded with null (0) selectors before entering the interrupt handler. This will permit the handler to safely save and restore the DS, ES, FS, and GS registers as 286 selectors. This is needed so that interrupt handlers which don't care about the mode of the interrupted program can use the same prolog and epilog code for state saving (i.e. push all registers in prolog, pop all in epilog) regardless of whether or not a "native" mode or Virtual 8086 mode program was interrupted. Restoring null selectors to these registers before executing the IRET will not cause a trap in the interrupt handler. Interrupt routines which expect values in the segment registers, or return values in segment registers will have to obtain/return values from the 8086 register images pushed onto the new stack. They will need to know the mode of the interrupted program in order to know where to find/return segment registers, and also to know how to interpret segment register values.

The IRET instruction will perform the inverse of the above sequence. Only the extended 386 IRET instruction (operand size = 32) can be used, and must be executed at level 0 to change the VM bit to 1.

(1) If the NT bit in the FLAGS register is on, an inter-task return is performed. The current state is stored in the current TSS, and the link field in the current TSS is used to locate the TSS for the interrupted task which is to be resumed.

Otherwise, continue with the following sequence.

(2) Read the FLAGS image from SS:8[ESP] into the FLAGS register. this will set VM to the value active in the interrupted routine.

(3) Pop off the instruction pointer CS:EIP. EIP is popped first, then a 32-bit word is popped which contains the CS value in the lower 16 bits. If VM=0, this CS load is done as a protected mode segment load. If VM=1, this will be done as an 8086 segment load.

(4) Increment the ESP register by 4 to bypass the FLAGS image which was "popped" in step 1.

(5) If VM=1, load segment registers ES, DS, FS, and GS from memory locations SS:[ESP+8], SS:[ESP+12], SS:[ESP+16], and SS:[ESP+20], respectively, where the new value of ESP stored in step 4 is used. Since VM=1, these are done as 8086 segment register loads.

Else if VM=0, check that the selectors in ES, DS, FS, and GS are valid in the interrupted routine. Null out invalid selectors to trap if an attempt is made to access through them.

(6) IF (RPL(CS) > CPL), pop the stack pointer SS:ESP from the stack. The ESP register is popped first, followed by 32-bits containing SS in the lower 16 bits. If VM=0, SS is loaded as a protected mode segment register load. If VM=1, an 8086 segment register load is used.

(7) Resume execution of the interrupted routine. The VM bit in the FLAGS register (restored from the interrupt routine's stack image in step 1) determines whether the processor resumes the interrupted routine in Protected mode or Virtual 8086 mode.

## 6. Instruction Set

The instruction set is compatible with the 86/186/286 family of processors. 86 and 186 programs can be executed in unprotected REAL mode, or as protected tasks within Virtual 8086 mode, a subset of protected mode. 286 programs can be executed in protected mode, and can be mixed with 32-bit code. The basic 86/186/286 instruction set has been extended in the following major ways:

(1) 2 segment registers have been added, along with instructions to manipulate them.

(2) A single-precision uncharacterized signed multiply has been added to eliminate one of the most frequently encountered characterizations, and to complement the IMUL Reg, Imm instruction added in the 186 and 286.

(3) Move with sign/zero extension instructions have been added for efficient data type conversion, particularly with the extended register set.

(4) A set of Conditional jumps with full displacements are added to eliminate the need to jump around jumps. This avoids pipeline breaks.

(5) A set of Byte Set on Condition instructions are added to set a byte to 0 or 1 depending on the setting of the condition flags.

(6) A complete set of bit instructions have been added.

(7) 32-bit extensions for operations and addressing are included.

The longest instruction is now 15 bytes long. The 2 operand arithmetic/logical Mem, Imm instructions can be 15 bytes long if all possible prefixes are used, along with the longest immediate field and effective address field. For example, if the "D" bit in the current segment is 0, indicating 16-bit defaults for address and data,

```
LOCK ADD FS:DWORD PTR [EBX+ESI*2+87654321H], 12345678H
```

would assemble as:

```
|66| |67| |64| |F0| |81| |84| |73| |87654321| |12345678|
```

An instruction longer than 15 bytes would have to have redundant prefixes such as more than 1 segment override, several LOCK prefixes, or a useless REP prefix. An attempt to execute an instruction longer than 15 bytes (with redundant prefixes) will raise a GP fault with 0 error code.

COMPATIBILITY NOTE:

> The 286 put a limit of 10 bytes on instruction length, so the 386 length of 15 is upwards compatible. The 8086 had no upper limit on the length of an instruction, so the 386 and 286 are incompatible with the 8086 in this regard.

## 6.1. 86 FAMILY EXTENSIONS

This section describes the instruction set extensions common to all future members of the 8086 family. In particular, all instructions in this section will be implemented both in the 386 and in the P8.

### 6.1.1. Segment Register Instructions

New instructions are added to operate with the new segment registers FS and GS.

(1) Two segment override prefixes

(2) LSS, LFS and LGS pointer load instructions (corresponding to LES and LDS)

(3) the reg field in the segment register move instruction is extended to include the new registers (there was an unused bit in that field that was specified to be zero)

(4) PUSH and POP segment register instructions are provided.

The segment override prefixes and the segment register move instructions will fit in the primary opcode map. The other instructions will be added in the escape map opened up by the

"2-byte" prefix in the 286 instruction set.

### 6.1.1.1. Segment Register Override Prefixes

Two new segment override prefixes are defined to permit the new segment registers to be used in forming addresses.

```
         +---------+
    FS:  |01100100|
         +---------+

         +---------+
    GS:  |01100101|
         +---------+
```

### 6.1.1.2. Segment Register Move Instructions

The segment register move instructions, opcodes 8C and 8E, have the interpretation of the "reg" field expanded so that 100 indicates FS, and 101 indicates GS. These two register codes were previously undefined; that is, the high order bit in the SReg field was always zero.

MOV SReg, R/M16

```
+---------+     +------------+
|10001110|     |mod|SReg|R/M|
+---------+     +------------+
```

The "SReg" field specifies the segment register to load. If SReg=001 (CS), 110, or 111 (undefined), an invalid opcode fault is raised.
If SReg=010 (SS), interrupts and debug traps/faults are|
inhibited at the end of the instruction. |

```
Sreg  := R/M16;
```

FLAGS AFFECTED: NONE.

MOV R/M16, SReg

```
+---------+     +------------+
|10001100|     |mod|SReg|R/M|
+---------+     +------------+
```

The "SReg" field specifies the segment register to store. If SReg=110, or 111, raise an invalid opcode fault.

```
R/M16 := SReg;
```

FLAGS AFFECTED: NONE.

Refer to chapter 2 for the encodings for the Segment registers used in the "SReg" field. "SReg" field codes of 110 and 111 will cause invalid opcode traps if used with these instructions. As before, "SReg" code field of 001 (CS) also causes an invalid opcode trap if used on the "load" form of these instructions.

### 6.1.1.3. Pointer Loads LSS, LFS, LGS

These instructions are added as the counterpart of the LES and LDS instructions. LFS and LGS are needed to provide pointer loads in single instructions for the new segment registers. LSS is provided so that a new stack pointer can be loaded in one indivisible instruction. As there is no room in the primary opcode map for these instructions, they are added in the 2-byte escape map opened up by the 0F primary opcode.

```
LSS  Reg, Mem
LFS  Reg, Mem
LGS  Reg, Mem

     +---------+     +---------+     +------------+
     |00001111|     |10110sss|     |mod|Reg|R/M|
     +---------+     +---------+     +------------+
```

The "mod" and R/M fields must specify a memory operand. If "mod" is 11, raise an invalid opcode fault.

16-bit operand size:

```
reg16 := mem16 @ EA;
Sreg  := mem16 @ EA+2;
```

32-bit operand size:

```
reg32 := mem32 @ EA;
Sreg  := mem16 @ EA+4;
```

FLAGS AFFECTED: NONE.

A full pointer is read from memory and stored into a segment register, general register pair. LSS loads the segment part into SS, LFS loads FS, and LGS loads GS. The "sss" field indicates the segment register (010=SS, 100=FS, 101=GS) to receive the segment part of the pointer. Other "sss" codes actually encode other instructions, and do not apply to this instruction form. A 32-bit operand size indicates a 48-bit segmented pointer load. A 16-bit operand size indicates a 32-bit segmented pointer load.

Unlike the other instructions which load SS, interrupts are not inhibited at the end of the LSS instruction. The other instructions (e.g. POP SS) inhibit interrupts to permit the following instruction to load SP to form an indivisible load of SS:SP. Since both SS and (E)SP are loaded by this instruction, the motivation for inhibiting interrupts is gone.

### 6.1.1.4. Push/Pop Segment Register

Instructions are defined to PUSH and POP either of the 2 new segment registers. Shorter 1 byte instructions are available for the original 4 segment registers. These instructions are not affected by the operand size prefix.

```
PUSH      sreg

     +--------+    +--------+
     |00001111|    |10sss000|
     +--------+    +--------+

POP       sreg

     +--------+    +--------+
     |00001111|    |10sss001|
     +--------+    +--------+
```

FLAGS AFFECTED: NONE.

The "sss" field indicates the segment register (100=FS, 101=GS) to push or pop. Other "sss" values are used to enocde other instructions.

IMPLEMENTATION NOTE:

> The encoding for these instructions was chosen to encode the segment register code in bits 5..3, just like the other segment register instructions.

### 6.1.2. Uncharacterized Multiply

An IMUL instruction that produces a single precision result and uses any register as the destination is defined, similar to the IMUL reg, imm instruction added in the 186 and 286. This instruction removes one of the more frequently encountered register characterizations in the 8086.

IMUL reg, R/M

```
+---------+     +---------+     +-----------+
|00001111|     |10101111|     |mod|Reg|R/M|
+---------+     +---------+     +-----------+
```

16-bit operand size:

```
tempreg16:reg16 := reg16 * R/M16;
if tempreg16 = sign extension(reg16)
     THEN CF, OF := 0;
     ELSE CF, OF := 1;
```

32-bit operand size:

```
tempreg32:reg32 := reg32 * R/M32;
if tempreg32 = sign extension(reg32)
     THEN CF, OF := 0;
     ELSE CF, OF := 1;
```

FLAGS AFFECTED: CF, OF : set on overflow
SF, ZF, AF, PF undefined

### 6.1.3. Move With Sign/Zero Extend

Instructions are added (in the "2-byte" escape map) to move an 8 or 16-bit register or memory value to a register with sign or zero extension to 16 or 32 bits.

```
MOVZX     Reg, R/M

+---------+     +---------+     +------------+
|00001111|      |1011011w|      |mod|Reg|R/M|
+---------+     +---------+     +------------+

16-bit operand size:

     IF w=0
          THEN Reg16 := ZERO EXTEND(Reg/Mem8);
          ELSE Reg16 := Reg/Mem16;  (* Normal Move *)

32-bit operand size:

     IF w=0
          THEN Reg32 := ZERO EXTEND(Reg/Mem8);
          ELSE Reg32 := ZERO EXTEND(Reg/Mem16);

FLAGS AFFECTED: NONE.
```

```
MOVSX     Reg, R/M

+---------+     +---------+     +------------+
|00001111|      |1011111w|      |mod|Reg|R/M|
+---------+     +---------+     +------------+

16-bit operand size:

     IF w=0
          THEN Reg16 := SIGN EXTEND(Reg/Mem8);
          ELSE Reg16 := Reg/Mem16;  (* Normal Move *)

32-bit operand size:

     IF w=0
          THEN Reg32 := SIGN EXTEND(Reg/Mem8);
          ELSE Reg32 := SIGN EXTEND(Reg/Mem16);

FLAGS AFFECTED: NONE.
```

The w bit indicates the size of the source operand (the R/M operand). w=0 indicates an 8-bit source. w=1 indicates a 16-bit source operand. The source operand is not affected by the operand size prefix, which is used only to indicate the size of the destination register. 16-bit operand size (8086) indicates a word destination register. 32-bit operand size (386) indicates a 32-bit destination register.

### 6.1.4. Long Displacement Conditional Jumps

A full set of conditional jumps are supplied which take a full sized displacement. A 16-bit operand size indicates a 16-bit displacement is present, the 16-bit IP is used (IP+DISP is stored into IP, with high order 16 bits of EIP cleared or undefined). A 32-bit operand size specifies a 32-bit displacement and the use of the full EIP register (EIP+DISP is stored into EIP).

```
Jcc     disp16/32

+---------+     +---------+     +-------------------------+
|00001111|      |1000tttn|      |   displacement16/32     |
+---------+     +---------+     +-------------------------+
```

```
IF tttn
     THEN IP := IP + Displacement16/32;
```

FLAGS AFFECTED: NONE.

where tttn is encoded as in the 8-bit displacement conditional jumps, as given in the table below.

### 6.1.5. BYTE Set on Condition

This group of instructions is provided to set a byte to 0 or 1 depending on any of the 16 conditions defined in the 8086. These are monadic operations, taking a single byte register or memory destination operand. The operand size prefix has no effect on this instruction. The format for SETcc is:

```
SETcc     r/m

+---------+     +---------+     +------------+
|00001111|      |1001tttn|      |mod|xxx|R/M|
+---------+     +---------+     +------------+
```

```
IF tttn
     THEN Reg/Mem8 : = 1;
     ELSE Reg/Mem8 : = 0;
```

FLAGS AFFECTED: NONE.

For the conditional instructions (conditional jumps and Set on Condition), tttn is encoded with n indicating to use the condition or its negation, and ttt giving the condition to test, as follows:

| tttn | Mnemonic | Meaning |
|---|---|---|
| 0000 | O | Overflow |
| 0001 | NO | No Overflow |
| 0010 | B | Below (unsigned) |
| | NAE | Not Above or Equal |
| 0011 | NB | Not Below |
| | AE | Above or Equal (unsigned) |
| 0100 | E | Equal |
| | Z | Zero |
| 0101 | NE | Not Equal |
| | NZ | Not Zero |
| 0110 | BE | Below or Equal |
| | NA | Not Above |
| 0111 | NBE | Not Below or Equal |
| | A | Above (unsigned) |
| 1000 | S | Sign |
| 1001 | NS | Not Sign |
| 1010 | P | Parity |
| | PE | Parity Even |
| 1011 | NP | Not Parity |
| | PO | Parity Odd |
| 1100 | L | Less than (signed) |
| | NGE | Not Greater or Equal |
| 1101 | NL | Not Less than |
| | GE | Greater or Equal |
| 1110 | LE | Less than or equal |
| | NG | Not Greater than |
| 1111 | NLE | Not less or Equal |
| | G | Greater than (signed) |

### 6.1.6. Single Bit Operations

This group of instructions operate on a single bit which can be in memory or in a register. If a register bit string is specified, the 16 or 32 bit register is implied, depending on the "operand size" of the instruction. For register bit strings the bit offset is taken MODULO the register size. For memory bit strings, bit number ("offset" MOD 8) within the byte at address ("base" + ("offset" DIV 8)) is selected.

The index of the selected bit can be given by an immediate constant in the instruction or by a value in a general register. Only an 8-bit immediate value can be specified in the instruction. This operand is taken modulo the operand size, so the range of immediate bit offsets is 0..31 (0..15 for 16-bit operand size). This allows any bit within a register to be selected. For memory bit strings, this immediate field gives only the bit offset within a word or dword. Immediate bit offsets larger than 31 (or 15) are supported by using the immediate bit offset field in combination with the displacement field of the memory operand. The low-order 3 to 5 bits of the immediate bit offset are stored in the immediate bit offset field, and the high order 27 to 29 bits are shifted and combined with the byte displacement in the addressing mode. The assembler will support arbitrary immediate values in these instructions, and will split these fields into the bit offset and memory displacement fields of the instruction.

If the bit offset operand is in a register, a full 32-bit (16-bit for 16-bit operand size) offset is accepted for memory bit strings, a 5 bit (4 bit for 16-bit operand size) offset for register bit strings.

These instructions first assign the value of the selected bit to CF, the carry flag. Then, for all instructions except BitTest, a new value is assigned to the selected bit, as determined by the operation.

PROGRAMMING NOTE:

> The processor is free to access the 4 bytes starting at the memory address EffectiveAddress + 4*(BitOffset DIV 32) if a 32-bit operand size is given (the 2 bytes at memory address EffectiveAddress + 2*(BitOffset DIV 16) for 16-bit operand size). Software must be careful when using these instructions for areas of memory close to address space holes. In particular, these instructions should not be used to reference memory mapped I/O registers directly. In this case it is recommended that software use MOV instructions to load from, or store to memory mapped device registers, and use the register forms of these instructions to manipulate the device register images loaded into processor registers.

### 6.1.6.1. Bit Test

```
BT   R/M, Reg                 offset=Reg

+---------+     +---------+    +------------+
|00001111|     |10100011|    |mod|Reg|R/M|
+---------+     +---------+    +------------+

BT   R/M, Imm8                offset=Imm8

+---------+     +---------+    +------------+   +---------+
|00001111|     |10111010|    |mod|100|R/M|   |  Imm8   |
+---------+     +---------+    +------------+   +---------+
```

```
          CF := Bit[R/M, offset]

     FLAGS AFFECTED: CF set to value of selected bit.
                     OF, SF, ZF, AF, PF are undefined
```

### 6.1.6.2. Bit Test and Set

```
BTS  R/M, Reg                 offset=Reg

+---------+     +---------+    +------------+
|00001111|     |10101011|    |mod|Reg|R/M|
+---------+     +---------+    +------------+

BTS  R/M, Imm8                offset=Imm8

+---------+     +---------+    +------------+   +---------+
|00001111|     |10111010|    |mod|101|R/M|   |  Imm8   |
+---------+     +---------+    +------------+   +---------+
```

```
          CF := Bit[R/M, offset]
          Bit[R/M, offset] := 1;

     FLAGS AFFECTED: CF set to value of selected bit.
                     OF, SF, ZF, AF, PF are undefined
```

### 6.1.6.3. Bit Test and Reset

```
BTR      R/M, Reg                offset=Reg

+---------+      +---------+      +------------+
|00001111|      |10110011|      |mod|Reg|R/M|
+---------+      +---------+      +------------+

BTR      R/M, Imm8               offset=Imm8

+---------+      +---------+      +------------+    +---------+
|00001111|      |10111010|      |mod|110|R/M|    |  Imm8   |
+---------+      +---------+      +------------+    +---------+
```

```
CF := Bit[R/M, offset]
Bit[R/M, offset] := 0;
```

FLAGS AFFECTED: CF set to value of selected bit.
OF, SF, ZF, AF, PF are undefined

### 6.1.6.4. Bit Test and Complement

```
BTC      R/M, Reg                offset=Reg

+---------+      +---------+      +------------+
|00001111|      |10111011|      |mod|Reg|R/M|
+---------+      +---------+      +------------+

BTC      R/M, Imm8               offset=Imm8

+---------+      +---------+      +------------+    +---------+
|00001111|      |10111010|      |mod|111|R/M|    |  Imm8   |
+---------+      +---------+      +------------+    +---------+
```

```
CF := Bit[R/M, offset]
Bit[R/M, offset] := NOT Bit[R/M, offset];
```

FLAGS AFFECTED: CF set to value of selected bit.
OF, SF, ZF, AF, PF are undefined

### 6.1.7. Double Shift Instructions

Operators to shift a double-precision quantity left or right to produce a single-precision value provide the basic hook needed to implement operations on long unaligned bit (or byte) strings.

The double shifts operate in a fashion similar to the existing shift instructions by shifting the R/M field by the count, but differ from the single width shifts by using the Reg operand to supply the bits to shift in, rather than shifting in 0's as the single width shifts do. In all cases, the R/M field specifies the input operand and the result location, the Reg field specifies an operand which supplies the bits to shift in, and the Imm or CL field gives the shift count. The R/M operand is shifted by the shift count, with bits from the Reg operand shifted in, and the result is stored in the R/M field; the contents of Reg remain unaltered. Because these operations specify the bits to shift in, rather than shifting in zeros, they operate as double width shifts. They are the basic operations needed to do a multiple precision shift, either a 64-bit shift, or a shift of an arbitrary length bit string. For 32-bit operands, the operations take a 64-bit input value formed by concatenating the Reg and R/M fields, and produce a 32-bit output. As with normal shifts, shift counts are taken MOD 32 to ensure a value in the range from 0 to 31.

For left shifts, the R/M field contains the high-order part of a double width input, with the Reg field holding the low-order part. The R/M field is shifted left, with high-order bits from the Reg operand shifted in on the right (low-order position). The result is stored back into the R/M operand.

For right shifts, the R/M field contains the low-order part of a double width input, with the Reg field holding the high-order part. The R/M field is shifted right, with low-order bits from the Reg operand shifted in on the left (high-order position). The result is stored back into the R/M operand.

### 6.1.7.1. Shift Left Double

```
SHLD   R/M, Reg, Imm8                        count=Imm8

+---------+      +---------+      +------------+   +---------+
|00001111|       |10100100|       |mod|Reg|R/M|    |  Imm8   |
+---------+      +---------+      +------------+   +---------+


SHLD   R/M, Reg, CL                          count=CL

+---------+      +---------+      +------------+
|00001111|       |10100101|       |mod|Reg|R/M|
+---------+      +---------+      +------------+
```

```
(* count  is an unsigned integer *)
(* Take count MOD 32 *)
shAmt := count MOD 32;
inBits := Reg; (* Allow overlapped operands *)
IF shAmt = 0
   THEN ; (* NOP *)
ELSE IF shAmt >= OperandSize
   THEN R/M, CF, OF, SF, ZF, AF, PF := UNDEFINED;
ELSE DO;
   CF := BIT[R/M, OperandSize-shAmt];
   FOR i := OperandSize-1 downto shAmt DO
       BIT[R/M, i] := BIT[R/M, i-shAmt];
   FOR i := shAmt-1 downto 0 DO
       BIT[R/M, i] := BIT[inBits, i-shAmt + OperandSize];
   IF BIT[R/M, OperandSize-1] <> CF
      THEN OF := 1;
      ELSE OF := 0;
   SetSZP(R/M);    (*SF, ZF, PF set acc. to result*)
   AF := UNDEFINED;
END;
```

FLAGS AFFECTED: CF set to value of last bit shifted out.
OF set if shifting last bit caused overflow.
SF, ZF, PF set according to result value.
AF undefined.

### 6.1.7.2. Shift Right Double

```
SHRD   R/M, Reg, Imm8                             count=Imm8

+---------+      +---------+      +------------+    +---------+
|00001111|       |10101100|       |mod|Reg|R/M|     |  Imm8   |
+---------+      +---------+      +------------+    +---------+


SHRD   R/M, Reg, CL                               count=CL

+---------+      +---------+      +------------+
|00001111|       |10101101|       |mod|Reg|R/M|
+---------+      +---------+      +------------+
```

```
(* count  is an unsigned integer *)
(* Take count MOD 32 *)
shAmt := count MOD 32;
inBits := Reg; (* Allow overlapped operands *)
IF shAmt = 0
   THEN ; (* NOP *)
ELSE IF shAmt >= OperandSize
   THEN R/M, CF, OF, SF, ZF, AF, PF := UNDEFINED;
ELSE DO;
   CF := BIT[R/M, shAmt-1];
   FOR i := 0 to OperandSize-1-shAmt DO
       BIT[R/M, i] := BIT[R/M, i+shAmt];
   FOR i := OperandSize-shAmt to OperandSize-1 DO
       BIT[R/M, i] := BIT[inBits, i+shAmt - OperandSize];
   IF BIT[R/M, OperandSize-1] <> BIT[R/M, OperandSize-2];
      THEN OF := 1;
      ELSE OF := 0;
   SetSZP(R/M);    (*SF, ZF, PF set acc. to result*)
   AF := UNDEFINED;
END;
```

FLAGS AFFECTED: CF set to value of last bit shifted out.
OF set if shifting last bit caused overflow.
SF, ZF, PF set according to result value.
AF undefined.

Example

One purpose of the double shifts is to implement a bit string move, with arbitrary misalignment of the two bit strings. A simple example is to move a bit string from an arbitrary offset into a word aligned byte string. A left-to-right string move works best if a double right shift is used inside the move loop.

```
    MOV  ESI, SrcAddr
    MOV  EDI, DestAddr
    MOV  EBX, dWordCnt
    MOV  CL, RelOffset        ; relative bit offset Dest-Src
    MOV  EDX, {ESI}           ; Load first DWORD of source
    ADD  ESI, 4               ; Bump source address
 BltLoop:
    LODS                      ; New high order part
    SHRD EDX, EAX, CL         ; EDX gets aligned DWORD.
    XCHG EDX, EAX             ; Swap high/low order parts.
    STOS                      ; Write out next aligned chunk.
    DEC  EBX
    JA   BltLoop
```

This loop is simple, yet allows the data to be moved in 32-bit pieces for the highest possible performance. Note that it assumes that the destination is aligned on a 32-bit boundary, and is an exact multiple of 32 bits long, but the source can be at an arbitrary bit boundary. A more general BitBlt would require some extra masking on the first DWORD moved (in front of the main loop) and on the last DWORD moved (after the main loop), but would have the same basic 32-bits per loop iteration as the code given above. Without a double wide shift, the best you can do is 16 bits per loop iteration by using a 32-bit shift, and replacing the XCHG with a ROT by 16 to swap high and low order parts of registers.

### 6.1.8. INSERT and EXTRACT

These instructions manipulate a sequence of bits. As with the single bit instructions, the bit strings can be either in a register or in memory. INSERT places the low order bits of a register into a bit string without disturbing the bits on either side of the inserted bits. EXTRACT extracts a bit string into a register where it is right justified and expanded with high-order zeros to fill the register.

These instructions take four operands: a base address for the bit string, the bit offset of the start of the substring to INSERT/EXTRACT, the length of the substring, and the register to obtain/deliver the value. The base is given as a register or memory address in the R/M field, the offset is contained in the EAX register (AX for 16-bit operand size), the field length is given in the CL register, and the Reg field gives the general register holding the value to INSERT or to hold the value EXTRACTED.

The bit offset is adjusted in one of two ways, depending on whether the bit string base is given as a register or a memory address. For register bit strings, the adjusted offset is obtained by taking the offset operand MODULO the operand size (16 or 32). For memory bit strings, the adjusted offset is obtained by taking the offset operand MODULO 8. Note that for memory bit strings, the effective address is adjusted by adding the offset operand DIV 8 to account for this bit offset adjustment. For memory bit strings, the processor is free to access the 4 bytes at memory address EffectiveAddress + BitOffset DIV 8 if a 32-bit operand size is given (the 2 bytes at EffectiveAddress + (BitOffset DIV 8) for 16-bit operand size).

If the bit length is larger than or equal to the operand size, or if the adjusted bit offset plus the bit length is larger than the operand size, the resulting value in the destination field is undefined. For EXTRACT, the destination field is given by the Reg operand. For INSERT, the destination field is given by the R/M operand. If the R/M field specifies a register (mod=11), the destination field is the specified register. If the R/M field specifies a memory address (mod=00, 01, or 10), the destination field is the 4-bytes at byte address EffectiveAddress + (BitOffset DIV 8) if the operand size is 32 (the 2 bytes at byte address EffectiveAddress + (BitOffset DIV 8) if the operand size is 16). In the out-of-range cases, the values stored in these locations are undefined, but no other memory locations or registers are affected.

PROGRAMMING NOTE:

> The processor is free to access the 4 bytes starting at the memory address EffectiveAddress + (BitOffset DIV 8) if a 32-bit operand size is given (the 2 bytes at memory address EffectiveAddress + (BitOffset DIV 8) for 16-bit operand

> size). Software must be careful when using these instructions for areas of memory close to address space holes. In particular, these instructions should not be used to reference memory mapped I/O registers directly. In this case it is recommended that software use MOV instructions to load from, or store to memory mapped device registers, and use the register forms of these instructions to manipulate the device register images loaded into processor registers.

INSERT with a length of 0 is a NOP. EXTRACT with a length of 0 moves a zero to the destination register.

## 6.1.8.1. INSERT base, offset, length, source

```
IBTS          R/M,    (E)AX,    CL,    Reg

+---------+    +---------+    +------------+    offset=(E)AX
|00001111|    |10100111|    |mod|Reg|R/M|
+---------+    +---------+    +------------+    length=CL
```

16-bit operand size:

```
(* offset  is signed;  length  is unsigned *)
IF   mod = 'b11 (*Register Destination*)
      AND (
           (offset+length) > 16
           OR offset < 0
           OR length >= 16                       |
          )
     THEN R/M16 := UNDEFINED;
ELSE IF   mod <> 'b11 (*Memory Destination*)
           AND (
                (offset MOD 8)+length) > 16
                OR length >= 16                  |
               )
      THEN WORD at (R/M+(offset DIV 8)):=UNDEFINED;

ELSE DO;
(* Source operands can overlap destination *)
count := length;  (* Field length taken from CL *)
temp := Reg;      (* Copy source *)
WHILE count > 0 DO
     count := count - 1;
     BIT[R/M, offset + count] := BIT[temp, count];
     END;
END;
```

32-bit operand size:

```
    (* offset  is signed;  length  is unsigned *)
    IF   mod = 'b11 (*Register Destination*)
          AND (
               (offset+length) > 32
               OR offset < 0
               OR length >= 32                        |
              )
         THEN R/M32 := UNDEFINED;
    ELSE IF   mod <> 'b11 (*Memory Destination*)
               AND (
                    (offset MOD 8)+length) > 32
                    OR length >= 32                   |
                   )
          THEN DWORD at (R/M+(offset DIV 8)):=UNDEFINED;

    ELSE DO;
    (* Source operands can overlap destination *)
    count := length;  (* Field length taken from CL *)
    temp := EReg;     (* Copy source *)
    WHILE count > 0 DO
         count := count - 1;
         BIT[R/M, offset + count] := BIT[temp, count];
         END;
    END;
```

FLAGS AFFECTED: OF, SF, ZF, AF, PF, CF UNDEFINED.

### 6.1.8.2. EXTRACT dest, base, offset, length

```
XBTS          Reg,  R/M, (E)AX ,  CL

+---------+    +---------+    +------------+    offset=(E)AX
|00001111|    |10100110|    |mod|Reg|R/M|
+---------+    +---------+    +------------+    length=CL
```

16-bit operand size:

```
(* offset  is signed;  length  is unsigned *)
IF   mod = 'b11 (*Register Destination*)
      AND (
           (offset+length) > 16
           OR offset < 0
           OR length >= 16                       |
          )
     THEN Reg := UNDEFINED;
ELSE IF   mod <> 'b11 (*Memory Destination*)
           AND (
                (offset MOD 8)+length) > 16
                OR length >= 16                  |
               )
     THEN Reg := UNDEFINED;

ELSE DO;
count := length; (* field length taken from CL *)
temp := 0;       (* Accumulate result in temp,
                 avoid src/dest overlap problems *)
WHILE count > 0 DO
     count := count - 1;
     BIT[temp, count] := BIT[R/M, offset + count];
     END;
Reg := temp;
END;
```

32-bit operand size:

```
(* offset  is signed;  length  is unsigned *)
IF   mod = 'b11 (*Register Destination*)
      AND (
           (offset+length) > 32
           OR offset < 0
           OR length >= 32                        |
          )
     THEN EReg := UNDEFINED;
ELSE IF   mod <> 'b11 (*Memory Destination*)
           AND (
                (offset MOD 8)+length) > 32
                OR length >= 32                   |
               )
     THEN EReg := UNDEFINED;

ELSE DO;
count := length; (* field length taken from CL *)
temp := 0;       (* Accumulate result in temp,
                 avoid src/dest overlap problems *)
WHILE count > 0 DO
     count := count - 1;
     BIT[temp, count] := BIT[R/M, offset + count];
     END;
EReg := temp;
END;
```

FLAGS AFFECTED: OF, SF, ZF, AF, PF, CF UNDEFINED.

### 6.1.9. ScanBit

These instructions scan a word (16 or 32 bits, depending on operand size) for a 1 bit, and store the index of the first set bit into the destination register. ScanBitForwards scans from right to left (starting with bit index 0 within the word). ScanBitReverse scans from left to right, starting with bit index 15 (for 16-bit operand sizes) or 31 (for 32-bit operand size). The ZF flag is set if the entire word is zero (no set bits are found), and reset if a 1 bit is found. If no set bit is found, the destination value is undefined.

#### 6.1.9.1. ScanBitForward Dest, Src

BSF Reg, R/M

```
+---------+    +---------+    +------------+
|00001111|    |10111100|    |mod|Reg|R/M|
+---------+    +---------+    +------------+
```

16-bit operand size:

```
(* Source is the 16-bit (WORD) R/M operand *)
(* Destination is the 16-bit (WORD) Reg operand *)
   IF R/M16 = 0
     THEN ZF := 1; Reg := UNDEFINED;
     ELSE
       temp := 0;
       ZF := 0;
       WHILE BIT[R/M16, temp] = 0 DO temp := temp+1;
       Reg := temp;
   FI
```

32-bit operand size:

```
(* Source is the 32-bit (DWORD) R/M operand *)
(* Destination is the 32-bit (DWORD) Reg operand *)
   IF R/M32 = 0
     THEN ZF := 1; EReg := UNDEFINED;
     ELSE
       temp := 0;
       ZF := 0;
       WHILE BIT[R/M32, temp] = 0 DO temp := temp+1;
       EReg := temp;
   FI
```

FLAGS AFFECTED: ZF.
OF, SF, AF, PF, CF UNDEFINED.

### 6.1.9.2. ScanBitReverse Dest, Src

BSR Reg, R/M

```
+---------+     +---------+     +-----------+
|00001111|     |10111101|     |mod|Reg|R/M|
+---------+     +---------+     +-----------+
```

16-bit operand size:

```
(* Source is the 16-bit (WORD) R/M operand *)
(* Destination is the 16-bit (WORD) Reg operand *)
   IF R/M16 = 0
     THEN ZF := 1; Reg := UNDEFINED;
     ELSE
       temp := 15;
       ZF := 0;
       WHILE BIT[R/M16, temp] = 0 DO temp := temp-1;
       Reg := temp;
   FI
```

32-bit operand size:

```
(* Source is the 32-bit (DWORD) R/M operand *)
(* Destination is the 32-bit (DWORD) Reg operand *)
   IF R/M32 = 0
     THEN ZF := 1; EReg := UNDEFINED;
     ELSE
       temp := 31;
       ZF := 0;
       WHILE BIT[R/M32, temp] = 0 DO temp := temp-1;
       EReg := temp;
   FI
```

FLAGS AFFECTED: ZF.
OF, SF, AF, PF, CF UNDEFINED.

## 6.2. 386 PROCESSOR SPECIFIC EXTENSIONS

This section describes the instructions that are added only to the 386 processor and its descendents. These will not be implemented in all members of the 8086 family, and so define a "branch" in the family tree. New instructions are required to load and store the Control and Test registers described in chapter 2. Support for the debug facilities in chapter 7 requires instructions to load and store the Debug Registers.

### 6.2.1. Load/Store Control Registers

```
MOV  CReg, Reg
MOV  Reg, CReg

+---------+      +---------+      +------------+
|00001111|      |001000d0|      |xx|CReg|Reg|
+---------+      +---------+      +------------+
```

FLAGS AFFECTED: OF, SF, ZF, AF, PF, CF UNDEFINED.

These instructions load and store values into the control registers. The "d" bit indicates the direction of movement: d=0 indicates that the value in the control register (CReg) is to be stored into a general register. d=1 indicates that the control register is to be loaded from a general register. The "mod" field is ignored for these instructions, which can take only register operands in the "r/m" field, as if the "mod" field were 11. These are privileged instructions. If executed at a privilege level other than 0, a protection fault is raised.

A 32-bit operand size is always used for these instructions, regardless of the default length setting or the presence of prefixes.

IMPLEMENTATION NOTE:
Only CR0, CR2, and CR3 are defined on the 386. Use of other registers will raise invalid opcode faults.

## 6.2.2. Load/Store Debug Registers

```
MOV  Dreg, Reg
MOV  Reg, Dreg

+---------+     +---------+     +------------+
|000011111|     |001000d1|     |11|DReg|Reg|
+---------+     +---------+     +------------+
```

FLAGS AFFECTED: OF, SF, ZF, AF, PF, CF UNDEFINED.

These instructions load and store values into the debug registers. The "d" bit indicates the direction of movement: d=0 indicates that the value in the debug register (DReg) is to be stored into a general register (Reg). d=1 indicates that the debug register is to be loaded from a general register (Reg). The "mod" field is ignored for these instructions, which can take only register operands in the "r/m" field, as if the "mod" field were 11.

PROGRAMMING NOTE:

> A value loaded into a debug register will change the operation of the processor in an implementation dependent way. In particular, the delay between loading a new value into a debug register, and the time that it takes effect is implementation dependent.

IMPLEMENTATION NOTE:

> All implementations must guarantee that new values loaded into the debug registers will take effect before the completion of the next IRET instruction. Implementations are free to have these values take effect sooner.

These are privileged instructions. If executed at a privilege level other than 0, a protection fault is raised.

A 32-bit operand size is always used for these instructions, regardless of the default length setting or the presence of prefixes.

IMPLEMENTATION NOTE:

> DR4 and DR5 are aliased to DR6 and DR7, respectively, by these load and store instructions.

### 6.2.3. Load/Store Test Registers

```
MOV  Treg, Reg
MOV  Reg, Treg

+---------+      +---------+      +------------+
|000011111|      |001001d0|      |11|TReg|Reg|
+---------+      +---------+      +------------+
```

These instructions load and store values into the test registers. The "d" bit indicates the direction of movement: d=0 indicates that the value in the test register (TReg) is to be stored into a general register (Reg). d=1 indicates that the test register is to be loaded from a general register (Reg). The "mod" field is ignored for these instructions, which can take only register operands in the "r/m" field, as if the "mod" field were 11.

The effect of loading or storing the Test registers are implementation defined.

These are privileged instructions. If executed at a privilege level other than 0, a protection fault is raised.

Only TR6(TReg=110) and TR7(TReg=111) are defined on the 386. Use of other codes in the TReg field of these instructions will have undefined results.

IMPLEMENTATION NOTE:

> Use of TR0..3 with these instructions will raise an undefined opcode fault (int6). Use of TR4 and TR5 will store precharge and load to nowhere.

## 6.3. INSTRUCTION ENHANCEMENT SUMMARY

The opcode encoding of the new instructions are designed to use bits in ways that the original 8086 and 186/286 instructions use the opcode bits. Bits 5..3 in the "primary" opcode byte encode operation extension values, and encode segment register numbers to match the convention used in the original instructions. The Bit TestXYZ instructions identify the function in bits 5..3 for both register bit offsets (operation extension in the "primary" opcode byte) and immediate bit offsets (operation extension in the MODRM byte). PUSH and POP FS/GS take the register number from bits 5..3, just like the 8086 instructions PUSH/POP ES/CS/SS/DS.

The new IMUL has the same code in bits 5..3 (101B) as the 8086 IMUL instruction. Bit 0 of the "opcode" is a 1, to match the byte/word interpretation of this bit in the original Multiply instructions.

MOVSX and MOVZX, along with the 8086 instruction CBW have bit 3 indicate whether to sign (bit 3 is 1) or zero (bit 3 is 0) extend. Bit 0 of the "opcode" indicates whether the source size is a byte (bit 0 is 0) or word (bit 0 is 1).

The codes for SHLD and SHRD were chosen to have the same codes in bits 5..3 as the 8086 SHL and SHR instructions.

| OPCODE | MNEMONIC | OPERANDS |
|---|---|---|
| 0F.20 | MOV | R/M, CReg |
| 0F.21 | MOV | Reg, DReg |
| 0F.22 | MOV | CReg, R/M |
| 0F.23 | MOV | DReg, Reg |
| 0F.24 | MOV | Reg, TReg |
| 0F.26 | MOV | TReg, Reg |
| 0F.80-F | Jcc | Disp16/32 |
| 0F.90-F | SETcc | R/M |
| 0F.A0 | PUSH | FS |
| 0F.A1 | POP | FS |
| 0F.A2 | | |
| 0F.A3 | BT | R/M, Reg |
| 0F.A4 | SHLD | R/M, Reg, imm8 |
| 0F.A5 | SHLD | R/M, Reg, CL |
| 0F.A6 | XBTS | Reg, R/M, EAX, CL |
| 0F.A7 | IBTS | R/M, EAX, CL, Reg |
| 0F.A8 | PUSH | GS |
| 0F.A9 | POP | GS |
| 0F.AA | | |
| 0F.AB | BTS | R/M, Reg |
| 0F.AC | SHRD | R/M, Reg, imm8 |
| 0F.AD | SHRD | R/M, Reg, CL |
| 0F.AE | | |
| 0F.AF | IMUL | Reg, R/M |
| 0F.B0 | | |
| 0F.B1 | | |
| 0F.B2 | LSS | Reg, Mem |
| 0F.B3 | BTR | R/M, Reg |
| 0F.B4 | LFS | Reg, Mem |
| 0F.B5 | LGS | Reg, Mem |
| 0F.B6 | MOVZX | Reg, R/M8 |
| 0F.B7 | MOVZX | Reg, R/M16 |
| 0F.B8 | | |
| 0F.B9 | | |
| 0F.BA.4 | BT | R/M, imm |
| 0F.BA.5 | BTS | R/M, imm |
| 0F.BA.6 | BTR | R/M, imm |
| 0F.BA.7 | BTC | R/M, imm |
| 0F.BB | BTC | R/M, Reg |
| 0F.BC | BSF | Reg, R/M |
| 0F.BD | BSR | Reg, R/M |
| 0F.BE | MOVSX | Reg, R/M8 |
| 0F.BF | MOVSX | Reg, R/M16 |
| 64 | FS: | ; Segment override prefix |
| 65 | GS: | ; Segment override prefix |

Where the BTxxx R/M, imm forms get their opcode from the "reg" bits in the following MODRM byte, as follows:

```
 7 6 5 4 3 2 1 0
+---+-----+-----+
|Mod| Opx | R/M |
+---+-----+-----+
```

| Opx | Function |
|---|---|
| 100 | BT |
| 101 | BTS |
| 110 | BTR |
| 111 | BTC |

## 6.4. 32-BIT EXTENSIONS

The 86/186/286 instruction set is extended in two orthogonal directions: 32-bit forms of all 16-bit instructions are added to support the 32-bit data types, and 32-bit addressing modes are made available for all instructions referencing memory. This orthogonal instruction set extension is accomplished by adding 2 prefixes to the instruction set. Their meaning depends on the setting of the D bit in the code segment descriptor (which gives the default length for operands and effective addresses when executing code contained in that segment) selected by the CS register. In the 8086 modes, no code segment descriptor is available, so D=0 (16-bit default) is used.

IF D = 0 (16-bit default lengths),

66 prefix indicates 32-bit operand lengths
67 prefix indicates 32-bit effective address

IF D = 1 (32-bit default lengths),

66 prefix indicates 16-bit operand lengths
67 prefix indicates 16-bit effective address

Code segment descriptors specify the defaults for the operand length and effective address size to permit better code density for sections of the program which are exclusively 32-bit code.

These 32-bit extensions are available in the 8086 modes (REAL or Virtual 8086 mode). In these modes the default is always 16-bit, so prefixes are needed to specify 32-bit operands or addresses.

These 32-bit extensions have a pervasive impact on the instruction set. In order to describe their effects, each large class of instructions will be discussed, with the effects of the 32-bit operand size and 32-bit effective address detailed. Appendix E summarizes these effects.

Unless specified otherwise, instructions with 8 and 16-bit operands do not affect the contents of the high-order 16 bits of the extended registers.

PROPRIETARY AND CONFIDENTIAL

## 6.4.1. DATA MOVEMENT INSTRUCTIONS

### 6.4.1.1. Move Instructions

| CODE | INSTRUCTION | OPERANDS | 32-OPERAND | 32-ADDRESS |
|---|---|---|---|---|
| 50-57 | PUSHw | reg | 1 | 0 |
| 58-5F | POPw | reg | 1 | 0 |
| 60 | PUSHA | | 1 | 0 |
| 61 | POPA | | 1 | 0 |
| 68 | PUSHw | imm16 | 1, 7 | 0 |
| 6A | PUSHw | imm8 | 1, 6 | 0 |
| 86 | XCHGb | R/M, Reg | 0 | 1 |
| 87 | XCHGw | R/M, Reg | 1 | 1 |
| 88 | MOVb | R/M, Reg | 0 | 1 |
| 89 | MOVw | R/M, Reg | 1 | 1 |
| 8A | MOVb | Reg, R/M | 0 | 1 |
| 8B | MOVw | Reg, R/M | 1 | 1 |
| 8D | LEAw | Reg, Mem | 1 | 1 |
| 8F | POPw | R/M | 1 | 1 |
| 90-97 | XCHGw | AX, reg | 1 | 0 |
| A0 | MOVb | AL, ADDR16 | 0 | 2 |
| A1 | MOVw | AX, ADDR16 | 1 | 2 |
| A2 | MOVb | ADDR16, AL | 0 | 2 |
| A3 | MOVw | ADDR16, AX | 1 | 2 |
| B0-B7 | MOVb | reg, imm8 | 0 | 0 |
| B8-BF | MOVw | reg, imm16 | 1, 7 | 0 |
| C6 | MOVb | R/M, imm8 | 0 | 1 |
| C7 | MOVw | R/M, imm16 | 1, 7 | 1 |
| FF.6 | PUSHw | R/M | 1 | 1 |

32-bit Operand Size Effects:

0. No effect
1. Word operands are 32 bits.
6. Immediate field is 8 bits, sign-extended to 32 bits.
7. Immediate field is 32 bits.

32-bit Effective Address size Effects:

0. No effect.
1. Memory addresses formed using 32-bit effective addresses.
2. Memory address given by 32-bit displacement only.

These instructions map fairly directly into 32-bit operations and 32-bit effective address modes. The LEA instruction has independent specification of the address and the destination size. A full 32-bit LEA requires both a 32-bit operand specification and a 32-bit effective address specification. Other combinations are:

| | |
|---|---|
| LEA reg16, EA16 | 286 LEA instruction |
| LEA reg32, EA16 | "convert" 286 EA to 386 offset. Effective address is truncated at 16 bits and stored in "reg32", extended with high order zeros. |
| LEA reg16, EA32 | "convert" 386 EA to 286 offset. Store low order 16 bits of EA in "reg16". |
| LEA reg32, EA32 | 32-bit load effective address |

### 6.4.1.2. Segment Register Instructions

| CODE | INSTRUCTION | OPERANDS | 32-OPERAND | 32-ADDRESS |
|---|---|---|---|---|
| 06 | PUSH | ES | 1 | 0 |
| 07 | POP | ES | 1 | 0 |
| 0E | PUSH | CS | 1 | 0 |
| 16 | PUSH | SS | 1 | 0 |
| 17 | POP | SS | 1 | 0 |
| 1E | PUSH | DS | 1 | 0 |
| 1F | POP | DS | 1 | 0 |
| 26 | ES: | Segment Override | 0 | 0 |
| 2E | CS: | Segment Override | 0 | 0 |
| 36 | SS: | Segment Override | 0 | 0 |
| 3E | DS: | Segment Override | 0 | 0 |
| 63 | ARPLw | R/M, Reg | 0 | 1 |
| 8C | MOVw | R/M, Sreg | 0 | 1 |
| 8E | MOVw | Sreg, R/M | 0 | 1 |
| C4 | LESw | Reg, Mem | 1, 2 | 1 |
| C5 | LDSw | Reg, Mem | 1, 2 | 1 |

32-bit Operand Size Effects:

0. No effect
1. Word operands are 32 bits.
2. Pointer operands are 48 bits.

32-bit Effective Address size Effects:

0. No effect.
1. Memory addresses formed using 32-bit effective addresses.

The instructions which specify a memory address can now take 32-bit effective addresses. The pointer load instructions will load a 48-bit pointer if they are preceded by a 32-bit operand size prefix.

PUSH/POP segment register increment/decrement the stack by 4 if a 32-bit operand size is given, in order to keep the stack aligned in 32-bit programs.

### 6.4.1.3. I/O Instructions

| CODE | INSTRUCTION | OPERANDS | 32-OPERAND | 32-ADDRESS |
|---|---|---|---|---|
| 6C | INSb | ES:DI, DX | 0 | 1, 3, 4 |
| 6D | INSw | ES:DI, DX | 1 | 1, 3, 4 |
| 6E | OUTSb | DX, DS:SI | 0 | 1, 3, 4 |
| 6F | OUTSw | DX, DS:SI | 1 | 1, 3, 4 |
| E4 | INb | AL, port8 | 0 | 0 |
| E5 | INw | AX, port8 | 1 | 0 |
| E6 | OUTb | port8, AL | 0 | 0 |
| E7 | OUTw | port8, AX | 1 | 0 |
| EC | INb | AL, DX | 0 | 3 |
| ED | INw | AX, DX | 1 | 3 |
| EE | OUTb | DX, AL | 0 | 3 |
| EF | OUTw | DX, AX | 1 | 3 |

32-bit Operand Size Effects:

0. No effect
1. Word operands are 32 bits.

32-bit Effective Address size Effects:

0. No effect.
1. Memory addresses formed using 32-bit effective addresses.
3. 16-bit port number taken from DX register.
   32-bit Effective Address size does not affect
   I/O port addresses.
4. String length (count) taken from 32-bit ECX register.

The I/O instructions map straightforwardly to 32-bit operations. The I/O space is not expanded beyond the 16 bits supported by the 86/186/286. When preceded by a 32-bit operand size prefix, the word I/O instructions will transfer 32-bits from the EAX register, or from the memory string addressed by ES:DI or DS:SI. The String I/O instructions can take a 32-bit effective address size prefix. In this case, the string address is formed using the 32-bit ESI or EDI register. Use of a 32-bit operand size prefix with a String I/O instruction causes the pointer register to auto-increment/decrement by 4.

The 64K I/O address space is mapped into the physical address space of the processor in an implementation defined manner. The I/O space maps directly to the physical space, rather than to the linear address space.

I/O cycles of 2 or 4 bytes to I/O addresses 0FFFD to 0FFFF will generate an I/O cycle to address 10000 (hex) rather than wrapping around.

To guarantee restartability, the string I/O instructions (INS and OUTS) must ensure that the memory address is accessible (no segment or page faults can occur) before running the I/O cycle.

### 6.4.1.4. Flags Instructions

| CODE | INSTRUCTION | OPERANDS | 32-OPERAND | 32-ADDRESS |
|---|---|---|---|---|
| 9C | PUSHF | | 22 | 0 |
| 9D | POPF | | 22 | 0 |
| 9E | SAHF | | 0 | 0 |
| 9F | LAHF | | 0 | 0 |
| F5 | CMC | | 0 | 0 |
| F8 | CLC | | 0 | 0 |
| F9 | STC | | 0 | 0 |
| FA | CLI | | 0 | 0 |
| FB | STI | | 0 | 0 |
| FC | CLD | | 0 | 0 |
| FD | STD | | 0 | 0 |

32-bit Operand Size Effects:

0. No effect
22. PUSH/POP 32-bit Extended FLAGS register.

32-bit Effective Address size Effects:

0. No effect.

The PUSHF and POPF instructions will push/pop the 32-bit extended FLAGS register if a 32-bit operand size prefix is used. The VM and RF bits are not loaded or stored by these instructions. PUSHF and POPF are IOPL sensitive instructions in Virtual 8086 mode.

## 6.4.2. ARITHMETIC/LOGICAL INSTRUCTIONS

### 6.4.2.1. Standard Arith/Logical Instructions

| CODE | INSTRUCTION | OPERANDS | 32-OPERAND | 32-ADDRESS |
|---|---|---|---|---|
| 00 | ADDb | R/M, Reg | 0 | 1 |
| 01 | ADDw | R/M, Reg | 1 | 1 |
| 02 | ADDb | Reg, R/M | 0 | 1 |
| 03 | ADDw | Reg, R/M | 1 | 1 |
| 04 | ADDb | AL, imm8 | 0 | 0 |
| 05 | ADDw | AX, imm16 | 1, 7 | 0 |
| 08 | ORb | R/M, Reg | 0 | 1 |
| 09 | ORw | R/M, Reg | 1 | 1 |
| 0A | ORb | Reg, R/M | 0 | 1 |
| 0B | ORw | Reg, R/M | 1 | 1 |
| 0C | ORb | AL, imm8 | 0 | 0 |
| 0D | ORw | AX, imm16 | 1, 7 | 0 |
| 10 | ADCb | R/M, Reg | 0 | 1 |
| 11 | ADCw | R/M, Reg | 1 | 1 |
| 12 | ADCb | Reg, R/M | 0 | 1 |
| 13 | ADCw | Reg, R/M | 1 | 1 |
| 14 | ADCb | AL, imm8 | 0 | 0 |
| 15 | ADCw | AX, imm16 | 1, 7 | 0 |
| 18 | SBBb | R/M, Reg | 0 | 1 |
| 19 | SBBw | R/M, Reg | 1 | 1 |
| 1A | SBBb | Reg, R/M | 0 | 1 |
| 1B | SBBw | Reg, R/M | 1 | 1 |
| 1C | SBBb | AL, imm8 | 0 | 0 |
| 1D | SBBw | AX, imm16 | 1, 7 | 0 |
| 20 | ANDb | R/M, Reg | 0 | 1 |
| 21 | ANDw | R/M, Reg | 1 | 1 |
| 22 | ANDb | Reg, R/M | 0 | 1 |
| 23 | ANDw | Reg, R/M | 1 | 1 |
| 24 | ANDb | AL, imm8 | 0 | 0 |
| 25 | ANDw | AX, imm16 | 1, 7 | 0 |
| 28 | SUBb | R/M, Reg | 0 | 1 |
| 29 | SUBw | R/M, Reg | 1 | 1 |
| 2A | SUBb | Reg, R/M | 0 | 1 |
| 2B | SUBw | Reg, R/M | 1 | 1 |
| 2C | SUBb | AL, imm8 | 0 | 0 |
| 2D | SUBw | AX, imm16 | 1, 7 | 0 |
| 30 | XORb | R/M, Reg | 0 | 1 |
| 31 | XORw | R/M, Reg | 1 | 1 |
| 32 | XORb | Reg, R/M | 0 | 1 |
| 33 | XORw | Reg, R/M | 1 | 1 |
| 34 | XORb | AL, imm8 | 0 | 0 |
| 35 | XORw | AX, imm16 | 1, 7 | 0 |
| 38 | CMPb | R/M, Reg | 0 | 1 |
| 39 | CMPw | R/M, Reg | 1 | 1 |
| 3A | CMPb | Reg, R/M | 0 | 1 |
| 3B | CMPw | Reg, R/M | 1 | 1 |

| CODE | INSTRUCTION | OPERANDS | 32-OPERAND | 32-ADDRESS |
|---|---|---|---|---|
| 3C | CMPb | AL, imm8 | 0 | 0 |
| 3D | CMPw | AX, imm16 | 1, 7 | 0 |
| 40-47 | INCw | reg | 1 | 0 |
| 48-4F | DECw | reg | 1 | 0 |
| 62 | BOUNDw | reg, R/M | 1 | 1 |
| 80 | IMMb | R/M, imm8 | 0 | 1 |
| 81 | IMMw | R/M, imm16 | 1, 7 | 1 |
| 82 | IMMb | R/M, imm8 | 0 | 1 |
| 83 | IMMw | R/M, imm8 | 1, 6 | 1 |
| 84 | TESTb | R/M, Reg | 0 | 1 |
| 85 | TESTw | R/M, Reg | 1 | 1 |
| A8 | TESTb | AL, imm8 | 0 | 0 |
| A9 | TESTw | AX, imm16 | 1,7 | 0 |
| C0 | SHIFTb | R/M, imm8 | 0 | 1 |
| C1 | SHIFTw | R/M, imm8 | 1 | 1 |
| D0 | SHIFTb | R/M, 1 | 0 | 1 |
| D1 | SHIFTw | R/M, 1 | 1 | 1 |
| D2 | SHIFTb | R/M, CL | 0 | 1 |
| D3 | SHIFTw | R/M, CL | 1 | 1 |
| F6.0 | TESTb | R/M, imm8 | 0 | 1 |
| F6.2 | NOTb | R/M | 0 | 1 |
| F6.3 | NEGb | R/M | 0 | 1 |
| F7.0 | TESTw | R/M, imm16 | 1, 7 | 1 |
| F7.2 | NOTw | R/M | 1 | 1 |
| F7.3 | NEGw | R/M | 1 | 1 |
| FE.0 | INCb | R/M | 0 | 1 |
| FE.1 | DECb | R/M | 0 | 1 |
| FF.0 | INCw | R/M | 1 | 1 |
| FF.1 | DECw | R/M | 1 | 1 |

32-bit Operand Size Effects:

0. No effect
1. Word operands are 32 bits.
6. Immediate field is 8 bits, sign-extended to 32 bits.
7. Immediate field is 32 bits.

32-bit Effective Address size Effects:

0. No effect.
1. Memory addresses formed using 32-bit effective addresses.

The standard arithmetic instructions map straightforwardly into 32-bit operations and 32-bit effective addresses. Word operands are taken as 32 bits if a 32-bit operand size is given. The BOUND instruction, if a 32 bit operand size is given, uses a 32-bit register and an 8-byte memory operand. The register value is compared against the two 4-byte bounds stored in memory.

All shifts and rotates take the shift count MOD 32, as in the 186 and 286. Because 32-bit shifts and rotates take the shift count MOD 32, they are limited to the range 0..31.

### 6.4.2.2. Multiply and Divide

| CODE | INSTRUCTION | OPERANDS | 32-OPERAND | 32-ADDRESS |
|---|---|---|---|---|
| 69 | IMULw | reg, R/M, imm16 | 1, 7 | 1 |
| 6B | IMULw | reg, R/M, imm8 | 1, 6 | 1 |
| F6.4 | MULb | AL, R/M | 0 | 1 |
| F6.5 | IMULb | AL, R/M | 0 | 1 |
| F6.6 | DIVb | AL, R/M | 0 | 1 |
| F6.7 | IDIVb | AL, R/M | 0 | 1 |
| F7.4 | MULw | AX, R/M | 16 | 1 |
| F7.5 | IMULw | AX, R/M | 16 | 1 |
| F7.6 | DIVw | AX, R/M | 17 | 1 |
| F7.7 | IDIVw | AX, R/M | 17 | 1 |

32-bit Operand Size Effects:

0. No effect
1. Word operands are 32 bits.
6. Immediate field is 8 bits, sign-extended to 32 bits.
7. Immediate field is 32 bits.
16. Double Precision multiply. 64-bit result stored in EDX:EAX register pair, formed by multiplying 32-bit EAX times 32-bit R/M value.
17. Double Precision divide. 64-bit dividend in EDX:EAX register pair is divided by 32-bit R/M value.
    32-bit Quotient stored in EAX register.
    32-bit Remainder stored in EDX register.

32-bit Effective Address size Effects:

0. No effect.
1. Memory addresses formed using 32-bit effective addresses.

Note that the left hand operand of the characterized multiplies and divides are implicit operands. They are not specified in the instructions, and are not given in the assembly language. Both operands must be specified for the non-characterized multiplies.

No additional divide instructions are supported on the 386. The new uncharacterized signed multiply was defined earlier in this chapter. The 386 supports the following forms of multiply:

```
MUL  R/M                             ; AX := AL * R/M byte
                                     ; DX:AX := AX * R/M word
                                     ; EDX:EAX := EAX * R/M dword
IMUL R/M                             ; AX := AL * R/M byte
                                     ; DX:AX := AX * R/M word
                                     ; EDX:EAX := EAX * R/M dword
IMUL Reg, R/M, imm                   ; Reg8 := R/M byte * imm
                                     ; Reg16 := R/M word * imm
                                     ; Reg32 := R/M dword * imm
IMUL Reg, R/M                        ; Reg8 := Reg8 * R/M byte
                                     ; Reg16 := Reg16 * R/M word
                                     ; Reg32 := Reg32 * R/M dword
```

The Byte multiplies and divides are not affected by operand size. The word multiplies and divides will operate with 32-bit words if a 32-bit operand size is given. The single precision IMULw operations will take 2 32-bit operands and produce a 32-bit result. The double precision multiply produces a 64-bit result in the EDX:EAX register pair. The CF and OF flags are set if EDX is not the sign extension of EAX; otherwise they are cleared. The double precision divide operates as indicated in note 17.

### 6.4.2.3. Conversion Instructions

| CODE | INSTRUCTION | OPERANDS | 32-OPERAND | 32-ADDRESS |
|---|---|---|---|---|
| 98 | CBWb | | 8 | 0 |
| 99 | CWDw | | 9 | 0 |

32-bit Operand Size Effects:

0. No effect
8. Sign extend value in AX, store result in EAX.
9. Sign extend value in EAX, store result in register pair EDX:EAX.

32-bit Effective Address size Effects:

0. No effect.

The CBWb instruction is the only "byte" instruction which is modified by the 32-bit operand size prefix. The convert instructions operate as follows:

```
16-bit CBW:
     AX := SIGN EXTENSION(AL);
32-bit CBW:
     EAX := SIGN EXTENSION(AX);

16-bit CWD:
     DX := SIGN(AX);
32-bit CWD:
     EDX := SIGN(EAX);
```

A 32-bit operand size given with the CBW instruction modifies both the "word" destination size and the "byte" source size. This permits the sign extension of a 16-bit word to 32-bits with one instruction. Extending a byte to 32 bits requires two instructions: a 16-bit CBW followed by a 32-bit CBW. Alternatively, one of the new MOVSx instructions can be used.

### 6.4.2.4. Decimal Arithmetic Instructions

| CODE | INSTRUCTION | OPERANDS | 32-OPERAND | 32-ADDRESS |
|---|---|---|---|---|
| 27 | DAA | 0 | 0 | |
| 2F | DAS | 0 | 0 | |
| 37 | AAA | 0 | 0 | |
| 3F | AAS | 0 | 0 | |
| D4 | AAM | 0 | 0 | |
| D5 | AAD | 0 | 0 | |

32-bit Operand Size Effects:

0. No effect

32-bit Effective Address size Effects:

0. No effect.

The Decimal instructions are unaffected by the 32-bit extensions.

### 6.4.2.5. Coprocessor Instructions

| CODE | INSTRUCTION | OPERANDS | 32-OPERAND | 32-ADDRESS |
|---|---|---|---|---|
| 9B | WAIT | | 0 | 0 |
| D8-DF | ESC | code, R/M | 21 | 1 |

32-bit Operand Size Effects:

0. No effect
21. Store 32-bit error information for FLDENV, FSTENV, FSAVE, and FRSTOR.

32-bit Effective Address size Effects:

0. No effect.
1. Memory addresses formed using 32-bit effective addresses.

Most Coprocessor instructions are not affected by the 32-bit operand size specification. Instructions which have a memory operand (ESC with a mod-R/M field indicating a memory operand) are affected by the 32-bit effective address specification in the usual fashion.

The instructions which load/store error information (FLDENV, FSTENV, FRSTOR, FSAVE) use the 16/32-bit operand size to determine the format of this error information. Use of the 16-bit operand size with these instructions uses the 16-bit 286/287 error format. Use of the 32-bit operand size with these instructions uses a new 32-bit format, defined below.

The 16-bit error format is shown below. The protected mode format is used if the 386 is operating in protected mode when the Load/Store ESC instruction is executed. The REAL mode format is used if the 386 is operating in REAL mode or Virtual 8086 mode when the Load/Store ESC instruction is executed.

```
   16-BIT PROTECTED MODE FORMAT
 15                             0
+--------------------------------+
|        CONTROL  WORD           | 0
+--------------------------------+
|         STATUS  WORD           | 2
+--------------------------------+
|          TAG    WORD           | 4
+--------------------------------+
|         IP    OFFSET           | 6
+--------------------------------+
|         CS  SELECTOR           | 8
+--------------------------------+
|     DATA OPERAND  OFFSET       | A
+--------------------------------+
|     DATA OPERAND SELECTOR      | C
+--------------------------------+
```

```
     16-BIT REAL MODE FORMAT
 15                             0
+--------------------------------+
|        CONTROL  WORD           | 0
+--------------------------------+
|         STATUS  WORD           | 2
+--------------------------------+
|          TAG    WORD           | 4
+--------------------------------+
|  INSTRUCTION POINTER 15..0     | 6
+-------+-+----------------------+
|IP19.16|0|    OPCODE 10..0      | 8
+-------+-+----------------------+
|     DATA POINTER  15..0        | A
+-------+------------------------+
|DP19.16|0 0 0 0 0 0 0 0 0 0 0 0|  C
+-------+------------------------+
```

The 32-bit mode formats are shown below. The protected mode format is used if the 386 is operating in protected mode. The REAL mode format is used if the 386 is operating in REAL mode or Virtual 8086 mode when the Load/Store ESC instruction is executed.

```
                 32-BIT   PROTECTED  MODE  FORMAT
 31                              15                            0
+-------------------------------+-------------------------------+
|       R E S E R V E D         |          CONTROL  WORD        |  0
+-------------------------------+-------------------------------+
|       R E S E R V E D         |           STATUS  WORD        |  4
+-------------------------------+-------------------------------+
|       R E S E R V E D         |            TAG    WORD        |  8
+-------------------------------+-------------------------------+
|                        IP    OFFSET                           |  C
+-------------------------------+-------------------------------+
|       R E S E R V E D         |           CS  SELECTOR        |  10
+-------------------------------+-------------------------------+
|                    DATA OPERAND  OFFSET                       |  14
+-------------------------------+-------------------------------+
|       R E S E R V E D         |      DATA OPERAND SELECTOR    |  18
+-------------------------------+-------------------------------+
```

```
                   32-BIT   REAL  MODE  FORMAT
 31                              15                            0
+-------------------------------+-------------------------------+
|       R E S E R V E D         |          CONTROL  WORD        |  0
+-------------------------------+-------------------------------+
|       R E S E R V E D         |           STATUS  WORD        |  4
+-------------------------------+-------------------------------+
|       R E S E R V E D         |            TAG    WORD        |  8
+-------------------------------+-------------------------------+
|       R E S E R V E D         |   INSTRUCTION POINTER 15..0   |  C
+-------+-----------------------+-------+-+---------------------+
|0 0 0 0|   INSTRUCTION POINTER 31..16  |0|     OPCODE  10..0   |  10
+-------+-----------------------+-------+-+---------------------+
|       R E S E R V E D         |      DATA     POINTER 15..0   |  14
+-------+-----------------------+-------+-----------------------+
|0 0 0 0|       DATA    POINTER 31..16  |0 0 0 0 0 0 0 0 0 0 0 0|  18
+-------+-----------------------+-------+-----------------------+
```

The fields marked RESERVED must be set to zero for FLDENV and FRSTOR, and the program must not be dependent on values stored in these fields by FSTENV and FSAVE.

## 6.4.3. STRING INSTRUCTIONS

| CODE | INSTRUCTION | OPERANDS | 32-OPERAND | 32-ADDRESS |
|---|---|---|---|---|
| A4 | MOVSb | ES:DI, DS:SI | 0 | 1 |
| A5 | MOVSw | ES:DI, DS:SI | 1, 14 | 1 |
| A6 | CMPSb | DS:SI, ES:DI | 0 | 1 |
| A7 | CMPSw | DS:SI, ES:DI | 1 | 1 |
| AA | STOSb | ES:DI, AL | 0 | 1 |
| AB | STOSw | ES:DI, AX | 1 | 1 |
| AC | LODSb | AL, DS:SI | 0 | 1 |
| AD | LODSw | AX, DS:SI | 1 | 1 |
| AE | SCASb | AL, ES:DI | 0 | 1 |
| AF | SCASw | AX, ES:DI | 1 | 1 |
| D7 | XLATb | AL, DS:[BX+AL] | 0 | 1 |
| F2 | REPNZw | CX | 0 | 4 |
| F3 | REPZw | CX | 0 | 4 |

32-bit Operand Size Effects:

0. No effect
1. Word operands are 32 bits.
14. Repeated String Move (REP MOVS) with 32-bit operand size can assume that its operands do not overlap destructively.

32-bit Effective Address size Effects:

0. No effect.
1. Memory addresses formed using 32-bit effective addresses.
4. String length (count) taken from 32-bit ECX register.

A 32-bit operand size specified with a string instruction indicates that the quantum of information is 4 bytes, so that data movement should take place in 4 byte chunks, and auto-increment/decrement should be by 4. A 32-bit effective address size specified with a string instruction indicates that the 32-bit ESI, EDI, and EBX (for XLAT) registers should be used for forming addresses, and that the 32-bit ECX register be used as the count register.

A 32-bit operand size on a repeated string move (MOVS) should be used only if the compiler (or programmer) can guarantee that the strings do not overlap destructively. If the strings overlap, the DF flag can be used to adjust the direction of movement to ensure that no destructive overlap occurs. If the strings overlap, the move will not overlap destructively provided that:

Addr(src) >= Addr(dest) and DF=0 (increment)
OR, Addr(src) <= Addr(dest) and DF=1 (decrement)

If the strings overlap destructively, and the destructive overlap must affect the move in a predictable way, the 8-bit or 16-bit string move should be used.

Implementation Note:

A straightforward implementation of a 16-bit string move where the destination address is odd and is 1 byte beyond (or before) the source address will not restart properly if a page fault is received when writing the destination. At the page break, the first byte of the word will be stored over the second byte of the source word at that step, so that when the instruction is restarted the source word fetched will be incorrect. The implementation must take care to ensure that MOVSw can restart properly. One way to do this is to probe unaligned writes to ensure that all of the individual write operations needed to perform an unaligned write can complete successfully before any of the writes are done.

### 6.4.4. CONTROL TRANSFERS

This section describes the instructions which alter the normal sequential execution order: Calls, Returns, Jumps, and Interrupts. One common aspect of these instructions is the size of the Instruction Pointer that participates in the operation. The mechanism for selecting between 16 and 32-bit operand sizes is used to indicate the use of the 16-bit IP register or the 32-bit EIP register. A 32-bit operand size indicates that the full 32-bit EIP register should be used, that displacements and new EIP values are 32-bits, and pointers are 48 bits. A 16-bit operand size indicates that the 86/186/286 compatible 16-bit IP is to be used, and that displacements and new IP values should be 16 bits, and code pointers should be 32 bits. 16-bit control transfers use 16-bit arithmetic to add displacements (if any), and store the result, or a new 16-bit displacement, into the IP register (low-order 16-bits of EIP register). After a 16-bit control transfer, the high-order bits of EIP are zeroed, so that predictable execution is guaranteed.

There is no prefix or default bit to indicate the size of IP to use when fetching instructions sequentially. The full 32-bit EIP register will always be used.

Programming Note:

> Because a 16-bit control transfer zeros the upper 16 bits of EIP, a 16-bit control transfer cannot safely transfer to, from, or within a segment larger than 64K. The processor does not enforce this restriction. Program development software (language translators and linkers) should enforce this restriction. In order to support transfers between old code (16-bit, code segments smaller than 64K) and new code (32-bit, large code segments), this restriction must be based on the actual size of the segment, not on the "D" bit of the segment (i.e. the "USE" attribute of the segment). An interface routine between new and old code requires the use of a code segment with the "D" bit 1, which is smaller than 64K, and which uses 16-bit control transfers to CALL old code.

Programming Note:

> Mixing 16-bit code with 32-bit code requires the resolution of a number of problems, only one of which is the IP size problem. The proper type of CALL must be used, so as to match the RETURN type used by a called routine. Some pointer translation may be necessary for parameter passing between 16 and 32-bit code. In order to handle the IP size problem, control transfers between new (32-bit) code in large segments, and old (16-bit) code in small segments (or vice-versa) must pass through a "buffer" routine which is contained in a code segment smaller than 64K, which is invoked (and returns) with one flavor of CALL/RETURN, and which invokes the other piece of code with the different

flavor of CALL/RETURN. It will likely have some pointer translation to do, as well as this CALL "translation" and IP size "conversion".

### 6.4.4.1. Procedure Call and Return

| CODE | INSTRUCTION | OPERANDS | 32-OPERAND | 32-ADDRESS |
|---|---|---|---|---|
| 9A | CALLf | immPtr | 2, 3, 12 | 0 |
| C2 | RETn | imm16 | 3, 11, 15 | 0 |
| C3 | RETn | | 3, 11 | 0 |
| C8 | ENTERw | framesize16, level8 | 15, 18 | 0 |
| C9 | LEAVEw | | 19 | 0 |
| CA | RETf | imm16 | 3, 10, 12, 15 | 0 |
| CB | RETf | | 3, 10, 12 | 0 |
| E8 | CALLn | disp16 | 3, 5 | 0 |
| FF.2 | CALLn | R/M (indirect) | 1, 3 | 1 |
| FF.3 | CALLf | R/M (indirect) | 2, 3, 12 | 1 |

32-bit Operand Size Effects:

0. No effect
1. Word operands are 32 bits.
2. Pointer operands are 48 bits.
3. Uses 32-bit EIP (Extended Instruction Pointer).
5. Displacement is 32 bits.
10. Inter-level RETurns pop a 48-bit SS:ESP old stack pointer from the stack. This pointer is padded to 64 bits with 16 undefined bits above the SS selector.
11. Pops a 32-bit EIP return offset from the stack.
12. Pushes/Pops a 48-bit CS:EIP return pointer from the stack. This pointer is padded to 64 bits with 16 undefined bits above the CS selector.
15. Immediate field is 16 bits, zero-extended to 32 bits.
18. 32-bit ENTER instruction. Uses 32-bit EBP, ESP registers. Pushes 32-bit display words from old display onto stack to form new display copy. 8-bit level number as in 286. Frame size is 16 bits zero extended to 32 bits.
19. 32-bit LEAVE instruction. Uses 32-bit EBP, ESP registers.

32-bit Effective Address size Effects:

0. No effect.
1. Memory addresses formed using 32-bit effective addresses.

The extension of pointer operands and offsets is controlled by the operand size selection so that the indirect CALLs can independently specify the effective address size for locating the indirect pointer, and the size of the indirect pointer.

The displacement field in the intra-segment call instruction is lengthened to 32-bits. No shorter form is provided.

The size of (E)SP used for the implicit stack references in CALLs, RETurns, ENTERs, and LEAVEs is discussed in chapter 3. Adding the immediate field given in a return and pop instruction is treated as an implicit stack reference. The size of (E)SP and (E)BP used in the explicit references to these registers in ENTER and LEAVE instructions is determined by the operand size. For ENTER, the operand size determines the size of each display element (2 or 4 bytes), the size of (E)BP pushed, and the size of the value to copy from (E)SP. For LEAVE, the operand size determines the size of (E)BP to copy to (E)SP. The addresses generated by ENTER and LEAVE are detemined by the "B" bit in the current SS descriptor.

Programming Note:

> Use of ENTER and LEAVE with operand size 16 but SS "B" bit 1 (32-bit setting), requires that the upper bits of both EBP and ESP be 0, since these instructions will copy only the lower 16 bits of these registers, but will use the full 32-bits of both registers to form addresses. Use of these instructions with operand size 32 and SS "B" bit 0 (16-bit setting) is well-defined for any values in the upper 16 bits of ESP and EBP, since these will copy the full 32-bit register values but use only the lower 16 bits for addressing. However, they will only be meaningful if the upper bits of the registers are 0 (indicating a stack less than 64K).

Rather than expanding the immediate fields in ENTER and RETurn from 16 to 32 bits, they can be kept at 16 bits, zero-extended to 32 bits. This will save 4 bytes per procedure over the alternative of expanding the fields to 32 bits.

Programming Note:

> If a procedure has more than 64K of parameters, the RETurn and pop instruction cannot be used. Instead, a plain RETurn instruction is used, and a SUB ESP, imm instruction is used after each call to the procedure. Similarly, if a local data frame is larger than 64K, an ENTER instruction with frame size of zero is used, followed by a SUB ESP, imm instruction.

Use of the 32-bit operand size prefix with a FAR CALL or RETurn instruction causes the selector pushed on the stack to be padded to 32 bits with high-order zeros to keep the stack aligned to a 32-bit boundary. This means that CS:EIP pointers are pushed as 64-bit quantities, with the 48-bit CS:EIP value in the lower 48 bits, and 16 upper bits undefined. Similarly, for interlevel calls and returns, the old stack pointer SS:ESP is pushed as a 64-bit quantity, with the 48-bit SS:ESP value in the low order 48 bits, and the 16 upper bits undefined.

### 6.4.4.2. Jump Instructions

| CODE | INSTRUCTION | OPERANDS | 32-OPERAND | 32-ADDRESS |
|---|---|---|---|---|
| 70-7F | Jcc | Disp8 | 3, 4 | 0 |
| E0 | LOOPNZ | CX, disp8 | 3, 4 | 4 |
| E1 | LOOPZ | CX, disp8 | 3, 4 | 4 |
| E2 | LOOP | CX, disp8 | 3, 4 | 4 |
| E3 | JCXZ | CX, disp8 | 3, 4 | 4 |
| E9 | JMPn | disp16 | 3, 5 | 0 |
| EA | JMPf | ptr32 | 2, 3 | 0 |
| EB | JMPn | disp8 | 3, 4 | 0 |
| FF.4 | JMPn | R/M (indirect) | 1, 3 | 1 |
| FF.5 | JMPf | R/M (indirect) | 2, 3 | 1 |

32-bit Operand Size Effects:

0. No effect
1. Word operands are 32 bits.
2. Pointer operands are 48 bits.
3. Uses 32-bit EIP (Extended Instruction Pointer).
4. Displacement is 8 bits, sign-extended to 32 bits.
5. Displacement is 32 bits.

32-bit Effective Address size Effects:

0. No effect.
1. Memory addresses formed using 32-bit effective addresses.
4. String length (count) taken from 32-bit ECX register.

In addition to the 32-bit effects discussed for CALL instructions, the jump instructions change the interpretation of 8-bit displacements so that they are sign-extended to 32-bits to obtain the actual displacement. Also, the 32-bit address size indicates that the 32-bit ECX register holds the count value for the LOOP and JCXZ instructions, to match the method used by the string instructions.

### 6.4.4.3. Interrupt Instructions

| CODE | INSTRUCTION | OPERANDS | 32-OPERAND | 32-ADDRESS |
|---|---|---|---|---|
| CC | INT | 3 | 13 | 0 |
| CD | INT | imm8 | 13 | 0 |
| CE | INTO | | 13 | 0 |
| CF | IRET | | 3,10,12,22 | 0 |

32-bit Operand Size Effects:

0. No effect
3. Uses 32-bit EIP (Extended Instruction Pointer).
10. Inter-level RETurns pop a 48-bit SS:ESP old stack pointer from the stack. This pointer is padded to 64 bits with 16 undefined bits above the SS selector.
12. Pushes/Pops a 48-bit CS:EIP return pointer from the stack. This pointer is padded to 64 bits with 16 undefined bits above the CS selector.
13. Operand size has no effect. (E)IP, (E)SP, and FLAGS size taken from Gate.
22. PUSH/POP 32-bit Extended FLAGS register.

32-bit Effective Address size Effects:

0. No effect.

IRET with a 32-bit operand size is used to return from an interrupt entered with one of the 386 interrupt gates (type 14 or 15). It undoes the actions performed at the trap. The ASM386 "equivalent" of IRET is:

```
     POP EIP   ; Old instruction pointer
     POP CS    ; 32-bit POP, selector in low 16 bits
     POPEF     ; Pop 32-bit Extended FLAGS register.
 ; If inter-level return, pop SS:ESP pointer
     POP ESP
     POP SS    ; 32-bit POP, selctor in low 16 bits.
 ; resume execution at popped CS:EIP
```

32-bit Interrupt gates will push the 32-bit extended FLAGS register, and will push 48-bit pointers for CS:EIP and SS:ESP. The 48-bit pointers are pushed as 64-bits, with the high-order 16 bits undefined. This will keep the stack aligned to a 32-bit boundary.

IRET with a 16-bit operand size is used to return from an interrupt entered with a 286 interrupt gate, or in REAL mode to return from an interrupt routine.

Either size IRET can be used to return from an interrupt handler invoked with a TASK gate (type 5). The NT flag (nested task) flag in the FLAGS word is referenced before anything is popped from the stack. If NT indicates an inter-task return, nothing is popped from the stack. Instead, the types of the TSSs involved in the task switch control whether a "386" or "286" return is performed.

The size of (E)SP used for the implicit stack references in Interrupts and IRETurns is discussed in chapter 3. All pushes and pops performed as part of interrupt and IRET processing are implicit stack references. The type of the gate (for interrupts) and the operand size (for IRET) determines the size of the SS:(E)SP pointer pushed onto/popped from the lower level stack, but does not impact the size of (E)SP used to generate the address for the push.

### 6.4.5. PROTECTION MODEL INSTRUCTIONS

| CODE | INSTRUCTION | OPERANDS | 32-OPERAND | 32-ADDRESS |
|---|---|---|---|---|
| 0F00.0 | SLDT | R/M | 0 | 1 |
| 0F00.1 | STR | R/M | 0 | 1 |
| 0F00.2 | LLDT | R/M | 0 | 1 |
| 0F00.3 | LTR | R/M | 0 | 1 |
| 0F00.4 | VERR | R/M | 0 | 1 |
| 0F00.5 | VERW | R/M | 0 | 1 |
| 0F01.0 | SGDT | R/M | 20 | 1 |
| 0F01.1 | SIDT | R/M | 20 | 1 |
| 0F01.2 | LGDT | R/M | 20 | 1 |
| 0F01.3 | LIDT | R/M | 20 | 1 |
| 0F01.4 | SMSW | R/M | 0 | 1 |
| 0F01.6 | LMSW | R/M | 0 | 1 |
| 0F02 | LAR | Reg,R/M | 1 | 1 |
| 0F03 | LSL | Reg,R/M | 1 | 1 |
| 0F06 | CLTS | | 0 | 0 |

32-bit Operand Size Effects:

0. No effect
1. Word operands are 32 bits.
20. Use all 6 bytes of the pseudo-descriptor.

32-bit Effective Address size Effects:

0. No effect.
1. Memory addresses formed using 32-bit effective addresses.

#### 6.4.5.1. LIDT, LGDT, SIDT, SGDT

The LIDT and LGDT instructions load a base address and limit value into the IDTR and GDTR, respectively. If a 16-bit operand size is given with this instruction, the register is loaded with a 16-bit limit and a 24 bit base, and the high-order 8 bits of the pseudo-descriptor are not used. If a 32-bit operand size is given with one of these instructions, a 16-bit limit and a 32-bit base is loaded using the high-order 8 bits of the pseudo-descriptor as high order base address bits.

The SIDT and SGDT instructions always store into all 48 bits of the pseudo-descriptor. On the 286, the upper 8 bits are are "undefined" after executing one of these instructions. On the 386, the upper 8 bits will be written with the high order 8 address bits, for both a 16-bit operand size and a 32-bit operand size. If the 16-bit operand size LIDT/LGDT instructions were used to load the register stored by SIDT/SGDT, the upper 8 bits are stored as zeros.

COMPATIBILITY NOTE:

The 16-bit forms of the LIDT/LGDT instructions are compatible with the 286. The 16-bit forms of the SIDT/SGDT instructions are compatible with the 286, provided that the value in the upper 8 bits is not referenced. The 286 will store 1s in these upper bits, whereas the 386 will store 0s if the 16-bit operand size is used. These bits were specified to be left undefined by the SIDT/SGDT instructions in the 286 Programmer's Reference Manual.

### 6.4.5.2. LAR

The LAR instruction stores a masked form of the second dword of the descriptor for the source selector, provided that the selector is visible at CPL (modified by the selector's RPL), and is a valid descriptor type. If the selector is invisible, or of the wrong type, ZF is cleared. Otherwise, the register is loaded with the high-order dword of the selector masked by 00FxFF00, and ZF is set to 1. The x indicates that the 4 bits corresponding to the upper 4-bits of the limit are undefined in the value loaded by LAR. If the 32-bit operand size is specified, the entire 32-bit value is loaded into the 32-bit destination register. If the 16-bit operand size is specified, the lower 16-bits of this value is stored into the 16-bit destination register. The 16-bit form of this instruction is exactly the same as the 286 LAR instruction.

Code and Data segment descriptors are valid for LAR. The special segment and gate descriptor types valid for LAR are given in the following table:

Descriptor Types for LAR

| code | type | valid |
|---|---|---|
| 0 | invalid | invalid |
| 1 | avail 286 TSS | valid |
| 2 | LDT | valid |
| 3 | busy 286 TSS | valid |
| 4 | 286 Call gate | valid |
| 5 | Task gate | valid |
| 6 | 286 Trap gate | valid |
| 7 | 286 Interrupt gate | valid |
| 8 | invalid | invalid |
| 9 | avail 386 TSS | valid |
| A | invalid | invalid |
| B | busy 386 TSS | valid |
| C | 386 Call gate | valid |
| D | invalid | invalid |
| E | 386 Trap gate | valid |
| F | 386 Interrupt gate | valid |

### 6.4.5.3. LSL

The LSL instruction loads a register with the unscrambled segment limit, and sets ZF to 1, provided that the source selector is visible at the CPL weakened by RPL, and the descriptor is a type accepted by LSL. Otherwise, ZF is cleared to 0, and the destination register is unchanged. The segment limit is loaded as a byte granular value. If the descriptor has a page granular segment limit, LSL will translate this to a byte limit before loading into the destination register (shift "raw" limit left 12 and or in low order 1's).

The 32-bit forms of this instruction store the 32-bit byte granular limit into the 32-bit destination register. The 16-bit forms store the low-order 16-bits of the byte granular limit into the 16-bit destination register.

Code and Data segment descriptors are valid for LSL. The special segment and gate descriptor types valid for LSL are given in the following table:

Descriptor Types for LSL

| code | type | valid |
|---|---|---|
| 0 | invalid | invalid |
| 1 | avail 286 TSS | valid |
| 2 | LDT | valid |
| 3 | busy 286 TSS | valid |
| 4 | 286 Call gate | invalid |
| 5 | Task gate | invalid |
| 6 | 286 Trap gate | invalid |
| 7 | 286 Interrupt gate | invalid |
| 8 | invalid | invalid |
| 9 | avail 386 TSS | valid |
| A | invalid | invalid |
| B | busy 386 TSS | valid |
| C | 386 Call gate | invalid |
| D | invalid | invalid |
| E | 386 Trap gate | invalid |
| F | 386 Interrupt gate | invalid |

### 6.4.6. MISCELLANEOUS INSTRUCTIONS

| CODE | INSTRUCTION | OPERANDS | 32-OPERAND | 32-ADDRESS |
|---|---|---|---|---|
| F0 | LOCK | 0 | 0 | |
| F4 | HLT | 0 | 0 | |

32-bit Operand Size Effects:

0. No effect

32-bit Effective Address size Effects:

0. No effect.

These instructions are not affected by the 32-bit extensions.

The LOCK prefix is guaranteed to work in all implementations of the architecture only for the following instructions:

| | |
|---|---|
| BitTestandSET/RESET/COMPLEMENT | Mem, Reg/imm |
| XCHG | Reg, Mem |
| XCHG | Mem, Reg |
| ADD, OR, ADC, SBB, AND, SUB, XOR | Mem, Reg/imm |
| NOT, NEG, INC, DEC | Mem |

An undefined opcode trap will be generated if a LOCK prefix is used with any instruction not listed above.

A LOCKed instruction is guaranteed to lock only the area of memory defined by the destination operand, but may lock a larger memory area. For example, typical 8086 and 286 configurations lock the entire memory space. This area of memory is guaranteed to be locked against access by a processor executing a LOCKed operation on exactly the same memory field (identical starting address, identical length). The integrity of the LOCK is not affected by the alignment of the memory field. Memory LOCKing is observed for arbitrarily misaligned fields.

LOCKed access is not guaranteed if another 386 processor concurrently executes an instruction that either:

- Is not preceded by a LOCK prefix, or
- Is not one of the instructions in the above list, or
- Specifies a memory operand that does not exactly overlap the destination operand. LOCKing is not guaranteed for partial overlap, even if one memory operand is wholly contained within another.

COMPATIBILITY NOTE:

Note that the 8086, the 186, and the 286 processors implement a superset of the LOCK function defined here. Programs using forms of LOCK specific to these processors may not execute properly if transported to a specific implementation of the 386.

The 286 does not interlock MOV instructions which operate with unaligned data. A MOV Mem, Reg instruction, where Mem is a 16-bit operand located at an odd address, will not assert the LOCK signal to LOCK the two bus cycles together.

## 7. Debug Support

In order to provide a basic level of debug support, debugging hooks are provided within the processor. The 86 family architecture defined two basic debug hooks: a one-byte trap and a single-step by instruction feature.

The 386 provides on-chip detection of breakpoint conditions. Up to 4 addresses can be monitored, with a debug exception raised if an instruction fetch, or a data read and/or write occurs at the given address.

A task switch trap is provided as a hook to permit software extension of the hardware task switch mechanism. In particular, it can be used to swap the debug facilities among a number of tasks. This trap is enabled by setting a bit in the TSS, and is described in chapter 3.

### 7.1. Terminology

The same terminology used in chapter 4 is used to describe the debug facilities. A Debug Exception is detected as part of executing an instruction, and so is synchronous with program execution. A Debug Condition is a generic term for a Debug Interrupt or a Debug Exception. A Debug Handler is the routine which receives control when a Debug Interrupt, or an enabled Debug Exception is detected.

A Debug Exception is classified as a fault or a trap depending on when the exception is raised. A Debug Trap is raised at the instruction boundary immediately after the execution of the instruction raising the exception, so that the CS:EIP value stored at entry to the exception handler points to the following instruction. A Debug Fault is raised at the instruction boundary immediately before the execution of the instruction raising the exception. At a fault, CS:EIP points to the instruction that raised the fault. Faults occur "before", Traps occur "after" the instruction raising the exception. The Task Switch and Single-step exceptions are traps. Instruction Breakpoints, and the Debug register protection exceptions are faults. Data Breakpoints are Traps. These are guaranteed to trap at the end of the breakpointed instruction.

COMPATIBILITY NOTE:

> The single-step feature on the 286 was incompatible with the 8086. The single-step mechanism had a number of problems on the 8086 which were "fixed" on the 286, resulting in incompatibilities. Single-stepping on the 386 is compatible with the 286.

## 7.2. Debug Registers

Six additional registers are added to control the Debug features. These registers can be accessed only at level 0, that is, the instructions which load and store these registers are privileged instructions. An optional level of protection is provided to fault to the debug handler (and turn off this check) if an attempt is made to load or store these registers. This protection is controlled by the BD/GD bits in DR6 and DR7, as described below.

```
 31                                                            0
+---------------------------------------------------------------+
|                                                               |
|                  Breakpoint 0 Linear Address                  |   DR0
|                                                               |
+---------------------------------------------------------------+
|                                                               |
|                  Breakpoint 1 Linear Address                  |   DR1
|                                                               |
+---------------------------------------------------------------+
|                                                               |
|                  Breakpoint 2 Linear Address                  |   DR2
|                                                               |
+---------------------------------------------------------------+
|                                                               |
|                  Breakpoint 3 Linear Address                  |   DR3
|                                                               |
+---------------------------------------------------------------+
|                                                               |
|                  IMPLEMENTATION DEFINED                       |   DR4
|                                                               |
+---------------------------------------------------------------+
|                                                               |
|                  IMPLEMENTATION DEFINED                       |   DR5
|                                                               |
+-------------------------------+-+-+-+-----------------+-+-+-+-+
|                               |B|B|B|                 |B|B|B|B|
|                               | | | |0 0 0 0 0 0 0 0 0| | | | |   DR6
|                               |T|S|D|                 |3|2|1|0|
+-------+-------+-------+-------+-+-+-+-----+-+-+-+-+-+-+-+-+-+-+
|LEN|R|W|LEN|R|W|LEN|R|W|LEN|R|W| | |G|     |G|L|G|L|G|L|G|L|G|L|
|       |       |       |       |0|0| |0 0 0| | | | | | | | | | |   DR7
|   3   |   2   |   1   |   0   | | |D|     |E|E|3|3|2|2|1|1|0|0|
+-------+-------+-------+-------+-+-+-+-----+-+-+-+-+-+-+-+-+-+-+
 3 3 2 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1 1
 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0
```

Debug Registers

DR0-DR3:
These registers contain the Linear Address associated with each of the four breakpoint conditions. Additional breakpoint qualifiers are given in DR7.

DR6: This register is the Debug Status register. The low-order bits are indicator bits which are set at entry to the debug exception handler when an enabled debug exception is detected. These bits are set even if a detected condition is not enabled by the mask bits in DR7. These bits are not set if a condition occurs which is masked by RF. These bits must be set accurately at instruction boundaries even if the GE/LE bits are not set.

IMPLEMENTATION NOTE:
The setting of condition bits into DR6 should occur after masking by RF (for debug faults), and before gating by the G/L bits to determine the need for a software break. Setting these bits accurately even if software breaks are not accurate (GE=LE=0) is easily done by sampling the conditions for inclusion into DR6 during the last clock (LC) of an instruction, or during first clock (FC) of the next instruction.

PROGRAMMING NOTE:
These bits are set by the hardware, but never cleared. Software should clear DR6 immediately before returning from the debug handler to avoid confusion in identifying the next exception.

DR7: This is the Debug Control register. The low order bits contain a mask to selectively enable the various debug conditions described below. The other fields are:

LEN|RWE0, LEN|RWE1, LEN|RWE2, LEN|RWE3:
These four 4-bit fields, one for each breakpoint register, provide the information needed to describe the break condition in addition to the linear address.

LEN: This 2-bit field indicates the length of the breakpoint field. If RWE=00 (instruction execution breakpoint), the breakpoint field must have a one byte length, that is, LEN=00, or the breakpoint actions are undefined. The linear address given for the breakpoint must be aligned on a multiple of the length of the breakpoint field. Unaligned items can be described by two or three breakpoint registers.

00 - 1 byte length
01 - 2 byte length
10 - UNDEFINED
11 - 4 byte length

Note that the complement of the LEN field can be used as a match mask on the low order bits of the breakpoint address to form the match field for the breakpoint.

RWE This 2-bit field indicates the type of access that should cause the breakpoint, encoded as follows:

| RWE | BREAK ON |
|---|---|
| 00 | Instruction Execution only. |
| 01 | Data Writes only |
| 10 | UNDEFINED |
| 11 | Data Reads or Writes only (not Instruction fetches) |

### 7.2.1. Debug Conditions

The low order bits of DR6 and DR7 are used to enable (DR7) and indicate (DR6) the various debug exceptions. The Bx bits in DR6 are set if the corresponding debug condition is active at an instruction boundary where an enabled debug condition occurs. These bits are set even if the condition is not enabled by the bits in DR7. The debug exception handler can access the Status register to see which breakpoint(s) were active when the exception handler was entered.

The Gx (Global enable) and Lx (Local enable) bits are used to enable selected debug conditions to cause the debug exception handler to be invoked. A given condition is enabled if either of its corresponding Gx or Lx bits is 1. That is, the Gx and Lx bits are ORed to obtain the enable status of condition x.

The Lx bits support fast task switching out of a task that uses some task-local breakpoint registers. They can be set while a task is executing, to enable one of the breakpoint register conditions. They are turned off by the processor at a task switch, to avoid spurious breakpoint exceptions in the new task.

The Gx bits enable global condtions such as the D condition, and also provide a global (i.e. not cleared at a task switch) enable bit for the breakpoint conditions. These bits support breakpoints that are active in all tasks in a system.

At each instruction boundary, the status of the debug conditions are formed into a vector corresponding to the Bx bits in DR6. The status of the debug conditions at an instruction boundary reflect the status of the debug interrupts and traps from the previous instruction, and the status of the debug faults from the next instruction masked by RF. The condition vector (not DR6) is then ANDed with the enable bits in DR7 (formed by ORing the Gx and Lx bits for each condition). If the result is non-zero, the condition vector is loaded into the Bx bits in DR6, and the debug exception handler is invoked.

The bit assignment is as follows:

BT: The BT bit is set if a task switch into a task occurs where the TSS for the new task has the DEBUG TRAP bit set. The DEBUG TRAP bit in the TSS is the enable bit for this condition, there is no enable bit in DR7.

BS: This bit is associated with the Single Step trap condition. The single-step condition is controlled by the TF bit in the EFLAGS register, as described in chapter 2. The BS bit in DR6 is set if the debug handler is entered due to the occurance of a single-step condition. This permits the debug handler to distinguish single-step traps from the other debug conditions.

Unlike the other bits in DR6, the BS bit is set only if a single-step trap actually occurs, not if a single-step condition (enabled or not) occurred.

PROGRAMMING NOTE:
The Single-Step by instruction is included in the set of debug conditions. This makes it difficult to use for debugging the debug exception handler. The one-byte trap instruction is the only easy to use debug tool for debugging the debug exception handler.

BD/GD:
The BD bit is set at an instruction boundary if the next instruction will read or write to one of the 8 Debug registers. The GD bit enables the Debug Register Protection condition. GD is cleared when the Debug exception handler is invoked to allow the debug handler free access to the debug registers.

GE/LE
Exact Data Breakpoint match required. If GE OR LE, slow processor execution to ensure that data breakpoints are reported on the instruction which causes them. If NOT(GE OR LE), bus cycles which occur "late" in an instruction may not be reported with the instruction causing the breakpoint, or may not be reported at all.

B0/G0/L0-B3/G3/L3:
These bits are used for the breakpoint conditions described by DR0-DR3 in conjunction with the high-order bits of DR7.

Bi is set if the condition described by the Linear Address in DRi, and the LENi/RWEi bits in DR7 occurs. Bi is set even if the condition is not enabled by Gi OR Li. Gi and Li are the Global and Local enable bits for breakpoint i. These are ORed together to mask the breakpoint condition. If Gi OR Li is 1, and if the ith breakpoint is detected, the processor will trap to the debug handler.

### 7.2.2. Breakpoint Address Recognition

The linear address and LEN field for each of the four breakpoint conditions define a range of sequential byte addresses for a breakpoint. The LEN field permits specification of a 1, 2, or 4 byte breakpoint field. Multiple-byte breakpoint fields must be aligned, that is, start at addresses that are a multiple of their length. This is enforced by ignoring low order bits in the breakpoint address register depending on the LEN field specified. LEN=00 indicates that all bits in the breakpoint address register are used as the breakpoint field start (and end) address. LEN=01 indicates that the high-order 31 bits in the breakpoint register are concatenated with a low-order zero bit to form the starting address of a two-byte breakpoint field. LEN=11 indicates that the high-order 30 bits in the breakpoint address register are concatenated with two low-order zeros to form the starting address of a four-byte breakpoint field.

Every data access and instruction fetch is checked against every breakpoint field address range. If a memory access occurs within the address ranges defined by any breakpoint address register and its LEN field, and the type of the access matches the RWE field for the breakpoint, the breakpoint condition is raised. Instruction fetch breakpoints need be signalled only if the first byte of an instruction matches a breakpoint address. As defined above, instruction fetch breakpoints must be explicitly defined as having a length of 1 byte (LEN=00) or else the breakpoint actions are undefined. Data write or read/write breakpoints must be triggered if any of the bytes participating in a memory access are within the breakpoint field defined by the breakpoint address register and its LEN field.

IMPLEMENTATION NOTE:

> Multiple byte memory references must be checked against multiple byte breakpoint fields. Care must be taken with unaligned memory references to ensure that this check is accurate. Multiple-byte breakpoint fields are required to be aligned so that checking these fields against arbitrary memory references is manageable.

## 7.3. System Errors

If interrupt 1 is fielded by a Task Gate, and the DEBUG TRAP bit is 1 in the TSS for the debug handler, an infinite loop of attempts to invoke the debug handler will result. Debug software should avoid setting the DEBUG TRAP bit in its own interrupt handler's TSS.

PROGRAMMING NOTE:

> We recommend that if the Debug Handler is fielded by a Task Gate, the TSS for the debug handler have the DEBUG TRAP bit set to 0.

## 7.4. Programming Notes

Breaks triggered by Data reads/writes are Traps, and so will occur at the instruction boundary "after" the instruction triggering the detection of the break condition. This means that a write into a location "protected" by a write breakpoint will be allowed to complete before the write breakpoint trap condition is reported. If a debugger wishes to preserve the contents of a breakpoint location, it can save the original contents before setting a Write breakpoint, and then copy this value back after the breakpoint is triggered. Since data breakpoints are traps, the breakpoint location can be referenced to obtain the new value stored by the operation which raised the breakpoint.

The RF flag permits the implementation of debug faults without requiring that resumption from a fault be implemented by disabling the fault condition, single stepping the next instruction, and then enabling the fault condition. Since single stepping loses control at interrupts and task switches (the TF bit in the flags register may be turned off), using single-stepping to resume from a debug fault would be very awkward.

The RF flag permits the debug handler to maintain an accurate count of debug fault occurances. After a debug fault, the debug handler can set the RF bit in the FLAGS image to be restored by the IRET instruction. The IRET will retry the faulting instruction with RF=1, so that debug faults will be ignored. If the retry of the instruction also raises other faults, the retry of the instruction after recovering from these faults will also be done with RF=1, so that debug faults are also ignored on subsequent retries. RF will be cleared after the successful execution of the faulting instruction.

A 16-bit IRET instruction will not affect the RF bit. REAL mode debuggers can use the 32-bit IRET after juggling the stack to have the appropriate values for the 32-bit EIP, CS, and EFLAGS, with the RF bit set to 1 in the EFLAGS value. Then the 32-bit IRET can be executed (requires an operand size prefix in REAL mode) to set the RF bit upon return from the debug handler.

## APPENDIX E: Summary of 32-bit Extensions

The following table summarizes the effect of specifying a 32-bit operand size or a 32-bit effective address with a 286 instruction. The 32-bit size can be specified either with a prefix or with the default size bit in the descriptor.

32-bit Operand Size Effects:

0. No effect
1. Word operands are 32 bits.
2. Pointer operands are 48 bits.
3. Uses 32-bit EIP (Extended Instruction Pointer).
4. Displacement is 8 bits, sign-extended to 32 bits.
5. Displacement is 32 bits.
6. Immediate field is 8 bits, sign-extended to 32 bits.
7. Immediate field is 32 bits.
8. Sign extend value in AX, store result in EAX.
9. Sign extend value in EAX, store result in register pair EDX:EAX.
10. Inter-level RETurns pop a 48-bit SS:ESP old stack pointer from the stack. This pointer is padded to 64 bits with 16 undefined bits above the SS selector.

11. Pops a 32-bit EIP return offset from the stack.
12. Pushes/Pops a 48-bit CS:EIP return pointer from the stack. This pointer is padded to 64 bits with 16 undefined bits above the CS selector.
13. Operand size has no effect. (E)IP, (E)SP, and FLAGS size taken from Gate.
14. Repeated String Move (REP MOVS) with 32-bit operand size can assume that its operands do not overlap.
15. Immediate field is 16 bits, zero-extended to 32 bits.
16. Double Precision multiply. 64-bit result stored in EDX:EAX register pair, formed by multiplying 32-bit EAX times 32-bit R/M value.
17. Double Precision divide. 64-bit dividend in EDX:EAX register pair is divided by 32-bit R/M value. 32-bit Quotient stored in EAX register. 32-bit Remainder stored in EDX register.
18. 32-bit ENTER instruction. Uses 32-bit EBP, ESP registers. Pushes 32-bit display words from old display onto stack to form new display copy. 8-bit level number as in 286. Frame size is 16 bits zero extended to 32 bits.
19. 32-bit LEAVE instruction. Uses 32-bit EBP, ESP registers.
20. Use all 6 bytes of the pseudo-descriptor.
21. Use 32-bit format for error information in FLDENV, FSTENV, FSAVE, and FRSTOR instructions.
22. PUSH/POP 32-bit Extended FLAGS register.
23. Bit offset taken from EAX register.

32-bit Effective Address size Effects:

0. No effect.
1. Memory addresses formed using 32-bit effective addresses.
2. Memory address given by 32-bit displacement only.
3. 16-bit port number taken from DX register.
   32-bit Effective Address size does not affect
   I/O port addresses.
4. String length (count) taken from 32-bit ECX register.

| CODE | INSTRUCTION | OPERANDS | 32-OPERAND | 32-ADDRESS |
|---|---|---|---|---|
| 00 | ADDb | R/M, Reg | 0 | 1 |
| 01 | ADDw | R/M, Reg | 1 | 1 |
| 02 | ADDb | Reg, R/M | 0 | 1 |
| 03 | ADDw | Reg, R/M | 1 | 1 |
| 04 | ADDb | AL, imm8 | 0 | 0 |
| 05 | ADDw | AX, imm16 | 1, 7 | 0 |
| 06 | PUSH | ES | 1 | 0 |
| 07 | POP | ES | 1 | 0 |
| 08 | ORb | R/M, Reg | 0 | 1 |
| 09 | ORw | R/M, Reg | 1 | 1 |
| 0A | ORb | Reg, R/M | 0 | 1 |
| 0B | ORw | Reg, R/M | 1 | 1 |
| 0C | ORb | AL, imm8 | 0 | 0 |
| 0D | ORw | AX, imm16 | 1, 7 | 0 |
| 0E | PUSH | CS | 1 | 0 |
| 0F00.0 | SLDT | R/M | 0 | 1 |
| 0F00.1 | STR | R/M | 0 | 1 |
| 0F00.2 | LLDT | R/M | 0 | 1 |
| 0F00.3 | LTR | R/M | 0 | 1 |
| 0F00.4 | VERR | R/M | 0 | 1 |
| 0F00.5 | VERW | R/M | 0 | 1 |
| 0F01.0 | SGDT | R/M | 20 | 1 |
| 0F01.1 | SIDT | R/M | 20 | 1 |
| 0F01.2 | LGDT | R/M | 20 | 1 |
| 0F01.3 | LIDT | R/M | 20 | 1 |
| 0F01.4 | SMSW | R/M | 0 | 1 |
| 0F01.6 | LMSW | R/M | 0 | 1 |
| 0F02 | LAR | Reg, R/M | 0 | 1 |
| 0F03 | LSL | Reg, R/M | 0 | 1 |
| 0F05 | | | | |
| 0F06 | CLTS | | 0 | 0 |
| 0F20 | MOV | R/M, CReg | 0 | 1 |
| 0F21 | MOV | R/M, DReg | 0 | 1 |
| 0F22 | MOV | CReg, R/M | 0 | 1 |
| 0F23 | MOV | DReg, R/M | 0 | 1 |
| 0F24 | MOV | R/M, TReg | 0 | 1 |
| 0F26 | MOV | TReg, R/M | 0 | 1 |
| 0F80-F | Jcc | disp16 | 5 | 0 |
| 0F90-F | Scc | R/M | 0 | 1 |

| | CODE | INSTRUCTION | OPERANDS | 32-OPERAND | 32-ADDRESS |
|---|---|---|---|---|---|
| 0FA0 | PUSH | FS | 1 | 0 | \| |
| 0FA1 | POP | FS | 1 | 0 | \| |
| 0FA3 | BT | R/M, Reg | 1 | 1 | |
| 0FA4 | SHLD | R/M, Reg, imm8 | 1 | 1 | \| |
| 0FA5 | SHLD | R/M, Reg, CL | 1 | 1 | \| |
| 0FA6 | XBTS | Reg, R/M, AX, CL | 1, 23 | 1 | |
| 0FA7 | IBTS | R/M, AX, CL, Reg | 1, 23 | 1 | |
| 0FA8 | PUSH | GS | 1 | 0 | \| |
| 0FA9 | POP | GS | 1 | 0 | \| |
| 0FAB | BTS | R/M, Reg | 1 | 1 | |
| 0FAC | SHRD | R/M, Reg, imm8 | 1 | 1 | \| |
| 0FAD | SHRD | R/M, Reg, CL | 1 | 1 | \| |
| 0FAF | IMUL | Reg, R/M | 1 | 1 | |
| 0FB2 | LSS | Reg, Mem | 1, 2 | 1 | |
| 0FB3 | BTR | R/M, Reg | 1 | 1 | |
| 0FB4 | LFS | Reg, Mem | 1, 2 | 1 | |
| 0FB5 | LGS | Reg, Mem | 1, 2 | 1 | |
| 0FB6 | MOVZX | Reg, R/M8 | 1 | 1 | |
| 0FB7 | MOVZX | Reg, R/M16 | 1 | 1 | |
| 0FBA.4 | BT | R/M, imm8 | 1 | 1 | |
| 0FBA.5 | BTS | R/M, imm8 | 1 | 1 | |
| 0FBA.6 | BTR | R/M, imm8 | 1 | 1 | |
| 0FBA.7 | BTC | R/M, imm8 | 1 | 1 | |
| 0FBB | BTC | R/M, Reg | 1 | 1 | |
| 0FBC | BSF | Reg, R/M | 1 | 1 | |
| 0FBD | BSR | Reg, R/M | 1 | 1 | |
| 0FBE | MOVSX | Reg, R/M8 | 1 | 1 | |
| 0FBF | MOVSX | Reg, R/M16 | 1 | 1 | |
| 10 | ADCb | R/M, Reg | 0 | 1 | |
| 11 | ADCw | R/M, Reg | 1 | 1 | |
| 12 | ADCb | Reg, R/M | 0 | 1 | |
| 13 | ADCw | Reg, R/M | 1 | 1 | |
| 14 | ADCb | AL, imm8 | 0 | 0 | |
| 15 | ADCw | AX, imm16 | 1, 7 | 0 | |
| 16 | PUSH | SS | 1 | 0 | \| |
| 17 | POP | SS | 1 | 0 | \| |
| 18 | SBBb | R/M, Reg | 0 | 1 | |
| 19 | SBBw | R/M, Reg | 1 | 1 | |
| 1A | SBBb | Reg, R/M | 0 | 1 | |
| 1B | SBBw | Reg, R/M | 1 | 1 | |
| 1C | SBBb | AL, imm8 | 0 | 0 | |
| 1D | SBBw | AX, imm16 | 1, 7 | 0 | |
| 1E | PUSH | DS | 1 | 0 | \| |
| 1F | POP | DS | 1 | 0 | \| |
| 20 | ANDb | R/M, Reg | 0 | 1 | |
| 21 | ANDw | R/M, Reg | 1 | 1 | |
| 22 | ANDb | Reg, R/M | 0 | 1 | |
| 23 | ANDw | Reg, R/M | 1 | 1 | |
| 24 | ANDb | AL, imm8 | 0 | 0 | |
| 25 | ANDw | AX, imm16 | 1, 7 | 0 | |
| 26 | ES: | Segment Override | 0 | 0 | |
| 27 | DAA | | 0 | 0 | |

| | CODE | INSTRUCTION | OPERANDS | 32-OPERAND | 32-ADDRESS |
|---|---|---|---|---|---|
| 28 | SUBb | R/M, Reg | 0 | 1 | |
| 29 | SUBw | R/M, Reg | 1 | 1 | |
| 2A | SUBb | Reg, R/M | 0 | 1 | |
| 2B | SUBw | Reg, R/M | 1 | 1 | |
| 2C | SUBb | AL, imm8 | 0 | 0 | |
| 2D | SUBw | AX, imm16 | 1, 7 | 0 | |
| 2E | CS: | Segment Override | 0 | 0 | |
| 2F | DAS | | 0 | 0 | |
| 30 | XORb | R/M, Reg | 0 | 1 | |
| 31 | XORw | R/M, Reg | 1 | 1 | |
| 32 | XORb | Reg, R/M | 0 | 1 | |
| 33 | XORw | Reg, R/M | 1 | 1 | |
| 34 | XORb | AL, imm8 | 0 | 0 | |
| 35 | XORw | AX, imm16 | 1, 7 | 0 | |
| 36 | SS: | Segment Override | 0 | 0 | |
| 37 | AAA | | 0 | 0 | |
| 38 | CMPb | R/M, Reg | 0 | 1 | |
| 39 | CMPw | R/M, Reg | 1 | 1 | |
| 3A | CMPb | Reg, R/M | 0 | 1 | |
| 3B | CMPw | Reg, R/M | 1 | 1 | |
| 3C | CMPb | AL, imm8 | 0 | 0 | |
| 3D | CMPw | AX, imm16 | 1, 7 | 0 | |
| 3E | DS: | Segment Override | 0 | 0 | |
| 3F | AAS | | 0 | 0 | |
| 40-47 | INCw | reg | 1 | 0 | |
| 48-4F | DECw | reg | 1 | 0 | |
| 50-57 | PUSHw | reg | 1 | 0 | |
| 58-5F | POPw | reg | 1 | 0 | |
| 60 | PUSHA | | 1 | 0 | |
| 61 | POPA | | 1 | 0 | |
| 62 | BOUNDw | reg, R/M | 1 | 1 | |
| 63 | ARPLw | R/M, reg | 0 | 1 | |
| 64 | FS: | Segment Override | 0 | 0 | |
| 65 | GS: | Segment Override | 0 | 0 | |
| 66 | DATA | SIZE Prefix | x | 0 | |
| 67 | ADDRESS | SIZE Prefix | 0 | x | |
| 68 | PUSHw | imm16 | 1, 7 | 0 | |
| 69 | IMULw | reg, R/M, imm16 | 1, 7 | 1 | |
| 6A | PUSHw | imm8 | 1, 6 | 0 | |
| 6B | IMULw | reg, R/M, imm8 | 1, 6 | 1 | |
| 6C | INSb | ES:DI, DX | 0 | 1,3,4 | |
| 6D | INSw | ES:DI, DX | 1 | 1,3,4 | |
| 6E | OUTSb | DX, DS:SI | 0 | 1,3,4 | |
| 6F | OUTSw | DX, DS:SI | 1 | 1,3,4 | |
| 70-7F | Jcc | Disp8 | 3, 4 | 0 | |
| 80 | IMMb | R/M, imm8 | 0 | 1 | |
| 81 | IMMw | R/M, imm16 | 1, 7 | 1 | |
| 82 | IMMb | R/M, imm8 | 0 | 1 | |
| 83 | IMMw | R/M, imm8 | 1, 6 | 1 | |
| 84 | TESTb | R/M, Reg | 0 | 1 | |
| 85 | TESTw | R/M, Reg | 1 | 1 | |
| 86 | XCHGb | R/M, Reg | 0 | 1 | |

| | CODE | INSTRUCTION | OPERANDS | 32-OPERAND | 32-ADDRESS |
|---|---|---|---|---|---|
| 87 | XCHGw | R/M, Reg | 1 | 1 | |
| 88 | MOVb | R/M, Reg | 0 | 1 | |
| 89 | MOVw | R/M, Reg | 1 | 1 | |
| 8A | MOVb | Reg, R/M | 0 | 1 | |
| 8B | MOVw | Reg, R/M | 1 | 1 | |
| 8C | MOVw | R/M, Sreg | 0 | 1 | |
| 8D | LEAw | Reg, Mem | 1 | 1 | |
| 8E | MOVw | Sreg, R/M | 0 | 1 | |
| 8F | POPw | R/M | 1 | 1 | |
| 90-97 | XCHGw | AX, reg | 1 | 0 | |
| 98 | CBWb | | 8 | 0 | |
| 99 | CWDw | | 9 | 0 | |
| 9A | CALLf | immPtr | 2, 3, 12 | 0 | |
| 9B | WAIT | | 0 | 0 | |
| 9C | PUSHF | | 22 | 0 | |
| 9D | POPF | | 22 | 0 | |
| 9E | SAHF | | 0 | 0 | |
| 9F | LAHF | | 0 | 0 | |
| A0 | MOVb | AL, ADDR16 | 0 | 2 | |
| A1 | MOVw | AX, ADDR16 | 1 | 2 | |
| A2 | MOVb | ADDR16, AL | 0 | 2 | |
| A3 | MOVw | ADDR16, AX | 1 | 2 | |
| A4 | MOVSb | ES:DI, DS:SI | 0 | 1 | |
| A5 | MOVSw | ES:DI, DS:SI | 1, 14 | 1 | |
| A6 | CMPSb | DS:SI, ES:DI | 0 | 1 | |
| A7 | CMPSw | DS:SI, ES:DI | 1 | 1 | |
| A8 | TESTb | AL, imm8 | 0 | 0 | |
| A9 | TESTw | AX, imm16 | 1, 7 | 0 | |
| AA | STOSb | ES:DI, AL | 0 | 1 | |
| AB | STOSw | ES:DI, AX | 1 | 1 | |
| AC | LODSb | AL, DS:SI | 0 | 1 | |
| AD | LODSw | AX, DS:SI | 1 | 1 | |
| AE | SCASb | AL, ES:DI | 0 | 1 | |
| AF | SCASw | AX, ES:DI | 1 | 1 | |
| B0-B7 | MOVb | reg, imm8 | 0 | 0 | |
| B8-BF | MOVw | reg, imm16 | 1, 7 | 0 | |
| C0 | SHIFTb | R/M, imm8 | 0 | 1 | |
| C1 | SHIFTw | R/M, imm8 | 1 | 1 | |
| C2 | RETn | imm16 | 3, 11, 15 | 0 | |
| C3 | RETn | | 3, 11 | 0 | |
| C4 | LESw | Reg, Mem | 1, 2 | 1 | |
| C5 | LDSw | Reg, Mem | 1, 2 | 1 | |
| C6 | MOVb | R/M, imm8 | 0 | 1 | |
| C7 | MOVw | R/M, imm16 | 1, 7 | 1 | |
| C8 | ENTERw | framesize16, level8 | 15, 18 | 0 | |
| C9 | LEAVEw | | 19 | 0 | |
| CA | RETf | imm16 | 3,10,12,15 | 0 | |
| CB | RETf | | 3,10,12 | 0 | |
| CC | INT | 3 | 13 | 0 | |
| CD | INT | imm8 | 13 | 0 | |
| CE | INTO | | 13 | 0 | |
| CF | IRET | | 3,10,12,22 | 0 | |

| | CODE | INSTRUCTION | OPERANDS | 32-OPERAND | 32-ADDRESS |
|---|---|---|---|---|---|
| D0 | SHIFTb | R/M, 1 | 0 | 1 | |
| D1 | SHIFTw | R/M, 1 | 1 | 1 | \| |
| D2 | SHIFTb | R/M, CL | 0 | 1 | \| |
| D3 | SHIFTw | R/M, CL | 1 | 1 | \| |
| D4 | AAM | | 0 | 0 | |
| D5 | AAD | | 0 | 0 | |
| D6 | | | | | |
| D7 | XLATb | AL, DS:[BX+AL] | 0 | 1 | |
| D8-DF | ESC | code, R/M | 21 | 1 | \| |
| E0 | LOOPNZ | CX, disp8 | 3, 4 | 4 | |
| E1 | LOOPZ | CX, disp8 | 3, 4 | 4 | |
| E2 | LOOP | CX, disp8 | 3, 4 | 4 | |
| E3 | JCXZ | CX, disp8 | 3, 4 | 4 | |
| E4 | INb | AL, port8 | 0 | 0 | |
| E5 | INw | AX, port8 | 1 | 0 | |
| E6 | OUTb | port8, AL | 0 | 0 | |
| E7 | OUTw | port8, AX | 1 | 0 | |
| E8 | CALLn | disp16 | 3, 5 | 0 | |
| E9 | JMPn | disp16 | 3, 5 | 0 | |
| EA | JMPf | ptr32 | 2, 3 | 0 | |
| EB | JMPn | disp8 | 3, 4 | 0 | |
| EC | INb | AL, DX | 0 | 3 | |
| ED | INw | AX, DX | 1 | 3 | |
| EE | OUTb | DX, AL | 0 | 3 | |
| EF | OUTw | DX, AX | 1 | 3 | |
| F0 | LOCK | | 0 | 0 | |
| F1 | | | | | |
| F2 | REPNZw | CX | 0 | 4 | |
| F3 | REPZw | CX | 0 | 4 | |
| F4 | HLT | | 0 | 0 | |
| F5 | CMC | | 0 | 0 | |
| F6.0 | TESTb | R/M, imm8 | 0 | 1 | |
| F6.2 | NOTb | R/M | 0 | 1 | |
| F6.3 | NEGb | R/M | 0 | 1 | |
| F6.4 | MULb | AL, R/M | 0 | 1 | |
| F6.5 | IMULb | AL, R/M | 0 | 1 | |
| F6.6 | DIVb | AL, R/M | 0 | 1 | |
| F6.7 | IDIVb | AL, R/M | 0 | 1 | |
| F7.0 | TESTw | R/M, imm16 | 1, 7 | 1 | |
| F7.2 | NOTw | R/M | 1 | 1 | |
| F7.3 | NEGw | R/M | 1 | 1 | |
| F7.4 | MULw | AX, R/M | 16 | 1 | |
| F7.5 | IMULw | AX, R/M | 16 | 1 | |
| F7.6 | DIVw | AX, R/M | 17 | 1 | |
| F7.7 | IDIVw | AX, R/M | 17 | 1 | |
| F8 | CLC | | 0 | 0 | |
| F9 | STC | | 0 | 0 | |
| FA | CLI | | 0 | 0 | |
| FB | STI | | 0 | 0 | |
| FC | CLD | | 0 | 0 | |
| FD | STD | | 0 | 0 | |
| FE.0 | INCb | R/M | 0 | 1 | |

|  | CODE | INSTRUCTION | OPERANDS | 32-OPERAND | 32-ADDRESS |
|---|---|---|---|---|---|
| FE.1 | DECb | R/M | 0 | 1 |  |
| FF.0 | INCw | R/M | 1 | 1 |  |
| FF.1 | DECw | R/M | 1 | 1 |  |
| FF.2 | CALLn | R/M | 1, 3 | 1 |  |
| FF.3 | CALLf | R/M | 2, 3, 12 | 1 |  |
| FF.4 | JMPn | R/M | 1, 3 | 1 |  |
| FF.5 | JMPf | R/M | 2, 3 | 1 |  |
| FF.6 | PUSHw | R/M | 1 | 1 |  |